VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Tvorba kolekce úloh pro výuku objektově orientovaného programování

Creation of the Problem Collection for Support of OOP Teaching

2013 **Pavel Rath**

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Pavel Rath**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Tvorba kolekce úloh pro výuku objektově orientovaného programování
Creation of the Problem Collection for Support of OOP Teaching

Zásady pro vypracování:

Cílem práce bude vytvořit sadu jednoduchých i pokročilejších úloh, které budou sloužit jako pomůcka při výuce předmětů, které se zabývají objektově orientovaným programováním a především JAVA technologií.

Postup, doplňující specifikace a další dílčí cíle:
1. Seznámit se s náplní předmětu "Programovací jazyky1".
2. Vymyslet sadu úloh, na kterých je možno procvičit problematiku z těchto oblastí:
a) objektové rysy jazyka,
b) kontejnery,
c) generické typy,
d) zpracování výjimek,
e) datové proudy,
f) paralelismus a souběžná vlákna provádění programu.
Pro každou oblast vznikne asi 10 úloh. U úloh je kladen důraz na to, aby výsledné řešení bylo snadno testovatelné.
3. Vytvořte jasnou strukturu zadání. Zadání by mělo obsahovat: specifikaci, omezení, obtížnost, tematicky zahrnuté oblasti, testovací data případně další věci.
4. Vytvořte ukázková řešení některých úloh.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Jan Kožusznik, Ph.D.**

Datum zadání: 16.11.2012

Datum odevzdání: 07.05.2013

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení studenta

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Dne: 25. 4. 2013

.....
podpis studenta

Poděkování

Velice rád bych poděkoval Ing. Janu Kožusznikovi, Ph.D. za odbornou pomoc, ochotu při konzultacích a věcné rady, které mi pomohly při vytváření této bakalářské práce. Dále bych chtěl poděkovat své rodině a přítelkyni za jejich podporu a pochopení.

Abstrakt

Tato bakalářská práce se zabývá vytvářením příkladů, podle kterých se lze učit objektově orientované programování. Konkrétně se jedná o technologie jazyka Java a jeho verzi Java SE. V první části práce jsou shrnuty některé teoretické znalosti potřebné pro vypracování jednotlivých zadání. Poté následují jednotlivá zadání, která jsou rozdělená do šesti kategorii. Jsou to objektové rysy jazyka, kontejnery, generické typy, zpracování výjimek, datové proudy a paralelismus s vlákny. Zadání jsou zaměřena na pokrytí jednotlivých technologií jazyka v dané kategorii a mají různou obtížnost. Každé zadání má jasně danou strukturu a obsahuje specifikaci, testování, omezení, obtížnost a zahrnuté oblasti. Ke každému zadání je vypracováno také ukázkové řešení.

Klíčová slova

Java, příklady pro výuku, kontejnery, generické typy, výjimky, datové proudy, vlákna.

Abstract

This bachelor thesis is about creating exercises which are designated for learning object-oriented programming. Specifically it's about Java technology and its version Java SE. In the first part of this thesis there are some theoretical knowledge that are needed for completing individual exercises. After that there are particular exercises which are divided into six groups. They are object features of language, collections, generic types, handling exceptions, data streams and parallelism with threads. Exercises are focused on covering each technology of language in specific category. Exercises have different difficulty. Each exercise have exact structure and contains specification, testing, limitation, difficulty and included areas. There is also finished sample solution for each exercise.

Key words

Java, exercises for learning, collections, generic types, exceptions, data streams, threads.

Seznam použitých zkratek

Zkratka	Anglický význam	Český význam
API	Application Programming Interface	Aplikační programové rozhraní
FIFO	First In, First Out	Metoda První dovnitř, první ven
GZIP	GNU zip	Souborový formát gzip
HTML	HyperText Markup Language	Značkovací jazyk pro hypertext
IP	Internet Protocol	Internetový protokol
JIT	Just-In-Time	Překlad v době potřeby
JVM	Java Virtual Machine	Virtuální stroj Javy
OOP	Object-oriented programming	Objektově orientované programování
URL	Uniform Resource Locator	Identifikátor umístění zdrojů
XML	Extensible Markup Language	Rozšiřitelný značkovací jazyk

Obsah

1 Úvod

Programovací jazyk Java byl oficiálně představen v roce 1995 a od té doby se těší velké popularitě mezi programátory. Jedná se o jazyk objektově orientovaný a mezi jeho největší výhody patří jeho přenositelnost. Java je neustále vyvíjena a nyní již existuje 7. verze. Za svou popularitu vděčí tomu, že lze v jednom jazyce napsat aplikace pro osobní počítače, mobilní telefony, internetové aplikace nebo třeba zařízení spotřební elektroniky. Pro specializaci na tyto různé cílové produkty existuje několik edicí Javy. Například Java SE je určená pro běžné počítačové aplikace, Java ME je se používá pro jednoduché mobilní aplikace, které byly velmi populární na starších mobilních telefonech. Další větší edicí je Java EE, která je určená pro velké "enterprise" projekty. V dnešní době se programovací jazyk Java používá také pro vývoj aplikací pro mobilní platformu Android.

Cílem této bakalářské práce je představit základní technologie edice Java SE a vytvořit sadu zadání, podle kterých bude možno tento jazyk vyučovat. Velký důraz je kladen na to, aby byla zadání přesně specifikována a jednoduše testovatelná. Zadání budou zaměřena na jednotlivé technologie a jejich použití, nikoliv na obecné programovací algoritmy. Jednotlivá zadání budou také obsahovat informaci o obtížnosti konkrétního zadání. Obtížnost je definována pomocí stupnice: 1 - začátečnická, 2 - mírně pokročilá, 3 - pokročilá, 4 - velmi pokročilá a 5 - těžká. Pro všechna zadání také bude vypracováno ukázkové řešení, které bude k nalezení v přílohách.

Na začátku této práce budou popsány teoretické základy Javy, objektové programování a poté technologie jednotlivých kategorií, ze kterých vzniknou zadání. V první skupině zadání budou představeny objektové prvky jako např. dědičnost, rozhraní, přetěžování a také dva návrhové vzory singleton a mediator. V druhé kategorii budou zadání orientované na ukládání objektů do různých kolekcí a také na vytvoření vlastní datové struktury jako je např. fronta nebo strom. V další kategorii budou představeny generické typy a využití jejich možností při programování. Následující kategorie bude obsahovat zadání pro procvičení vyvolávání a odchytávání výjimek jak předdefinovaných, tak vlastních. Pátá kategorie bude zaměřena na práci s datovými proudy a v jednotlivých zadáních budou procvičeny různé typy ukládání do souboru a jejich čtení a také např. aplikace Server-Klient. Poslední kategorie bude sloužit k vyzkoušení si práce s vlákny a ošetření problémů s vývojem paralelních aplikací.

2 Teoretické znalosti

2.1 Java jako programovací jazyk

Java je objektově orientovaný programovací jazyk, který vymysleli v roce 1991 James Gosling, Patrick Naughton, Chris Warth, Ed Frank a Mike Sheridan ze společnosti Sun Microsystems. Původně se jazyk nazýval Oak, ale v roce 1995 byl přejmenován na Java. Nynějším vlastníkem Javy, který dále jazyk rozvíjí, je společnost Oracle Corporation. Oracle získal Javu od firmy Sun v roce 2010. [5]

Hlavní motivací pro vznik jazyka byla potřeba programovacího jazyka, který by nebyl závislý na platformě a bylo by možné stejný program spouštět na osobních počítačích s jakýmkoliv operačním systémem stejně dobře jako např. na zařízeních spotřební elektroniky - set-top box, mikrovlnná trouba a jiné.

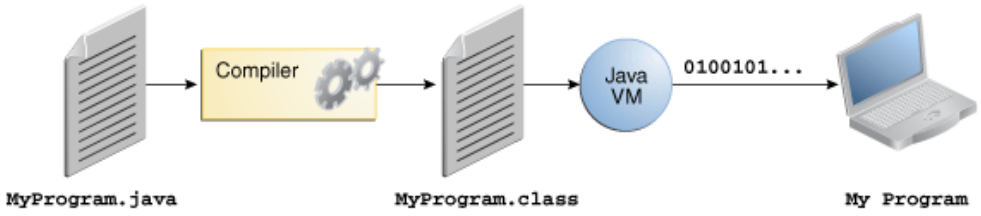
2.2 Java jako platforma

Přenositelnost aplikací byla docílena pomocí vlastní Java platformy. Ta se skládá ze dvou částí:

- aplikační programové rozhraní - Java Application Programming Interface (API)
- virtuální stroj - Java Virtual Machine (JVM)

API je v podstatě velká kolekce předem připravených programových komponent, které poskytují spoustu užitečných funkcí. Všechny tyhle funkce jsou zabaleny do knihoven obsahujících odpovídající třídy a rozhraní. Tyto knihovny jsou známé jako balíčky (packages). [7]

Virtuální stroj Javy je dostupný v několika verzích pro daný systém a provádí samotný program, který ale již v implementaci nezávisí na daném systému. Tento způsob spouštění také zvyšuje bezpečnost, protože vše řídí virtuální stroj a může programu zamezit v provádění vedlejších efektů mimo daný systém. [5] Celý proces spouštění je následující: programátor napíše zdrojový kód uložený v souborech s příponou *.java*. Následně je zdrojový kód zkompilován do tzv. bajtového kódu (bytecode), který se uloží do souboru *.class* (viz. Obrázek 2.1). Tento bajtový kód JVM následně interpretuje.



Obrázek 2 .1: Kompilace programu [7]

"Program při interpretování obecně běží pomaleji, než jak by běžel stejný program při zkompilování spustitelného kódu. Nicméně v případě Javy není tento rozdíl zase tak velký. Bajtový kód je vysoce optimalizovaný, což umožňuje virtuálnímu stroji Javy, aby prováděl programy mnohem rychleji." [5] Pro zvýšení výkonu se nedlouho po prvním vydání Javy začal používat tzv. JIT

kompilátor (Just-In-Time), který za běhu kompiluje vybrané části bajtového kódu do spustitelného kódu podle toho, jak jsou vyžadovány. [5]

2.3 Objektově orientované programování

Objektově orientované programování je metoda programování, která vychází z myšlenky, že všechny programy, jež vytváříme jsou simulací skutečného nebo námi vymyšleného světa. Všechny tyto světy jsou světy objektů, které mezi sebou komunikují (navzájem si posílají zprávy). [4] Každý objekt je instancí určité třídy. Třída je tedy jednotná šablona pro objekty stejného typu, které vytváříme. Jednotlivé objekty jsou definovány třemi částmi:

- identitou - svým názvem,
- daty - svými atributy,
- chováním - svými metodami.

2.3.1 Zapouzdření

"Zapouzdření je programovací mechanizmus, jenž svazuje kód s daty, s nimiž pracuje, a který udržuje kód i data v bezpečí před narušením a nesprávným použitím zvenčí." [5] Data i metody lze označit jako soukromé nebo veřejné. K soukromým částem lze přistupovat pouze v rámci dané třídy. K veřejným prvkům můžeme přistupovat i z "vnějšího kódu" mimo danou třídu. Další možností je označit data nebo metody jako chráněné. K takovým prvkům lze poté přistupovat buď z dané třídy nebo z třídy zděděné.

2.3.2 Dědičnost

"Dědičnost je proces, kdy jeden objekt získává vlastnosti jiného objektu. To je důležité, protože podporuje princip hierarchické klasifikace." [5] Jako příklad si můžeme uvést třídu `Strom`, která by představovala společné vlastnosti a atributy pro všechny stromy. Z této třídy by poté mohly dědit např. `ListnatyStrom` a `JehlicnatyStrom`. Každá zděděná třída obsahuje prvky třídy rodičovské a navíc obsahuje své specifické vlastnosti. Výhodou tohoto procesu je, že nemusíme všechny společné atributy a metody definovat znovu v každé třídě, ale pouze jednou v třídě rodičovské.

Java podporuje pouze jednoduchou dědičnost, což znamená, že každá třída může mít pouze jednoho přímého předka - může dědit pouze z jedné třídy.

2.3.3 Polymorfizmus

"Slovo polymorfizmus lze nejblíže vyjádřit českým slovem vícetvarost nebo mnohotvarost. Jedná se o možnost využívat v programovém textu stejnou syntaktickou podobu pro metody s různou vnitřní reprezentací." [3] Základním principem polymorfizmu je, že můžeme do proměnné datového typu předka uložit potomka nebo také do proměnné datového typu rozhraní uložit třídu, která dané rozhraní implementuje. Poté se až za běhu programu určí z jakého objektu se bude metoda volat.

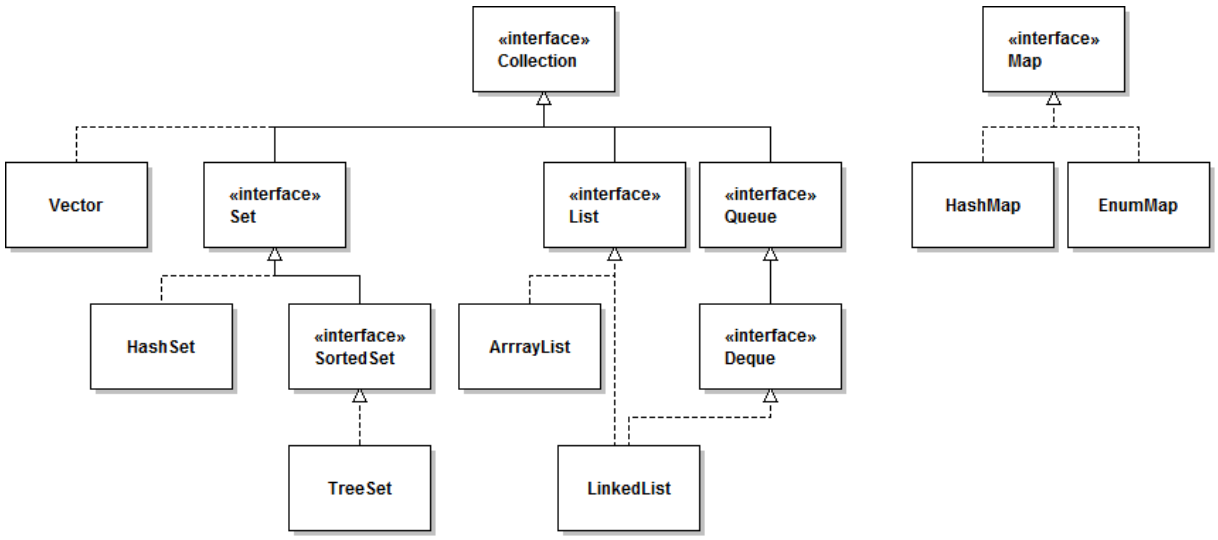
Polymorfizmus tedy souvisí i s dědičností. V předkovi nadefinujeme nějakou metodu, kterou můžeme v potomkovi přepsat jinou implementací. Metodu můžeme v předkovi také označit jako abstraktní, což nám určuje, že tělo metody bude definované až v potomcích. Další možností je také

využití rozhraní, ve kterém pouze nadefinujeme hlavičku metody a o tělo metody se starají třídy, které rozhraní implementují.

2.4 Kontejnery

Kontejnery jsou také někdy označované jako kolekce a jsou to objekty tříd Java API z balíku `java.util`. Kontejnery slouží pro uchovávání několika objektů, jejichž počet není předem známý. [2] Jedná se v podstatě o objekty, které uchovávají další objekty. Mezi výhody kolekcí patří fakt, že snižují množství potřebného kódu, protože většina potřebných algoritmů pro práci s daty je již naprogramována. Také zvyšují přehlednost kódu, protože nemusíme definovat vlastní datové struktury. Výhodou oproti klasickým polím je, že nemusíme předem definovat počet uložených prvků a přistupovat k nim pouze na základě číselného indexu. Nevýhodou pak je, že nelze ukládat primitivní typy, např. `int`. Tyto prvky musíme ukládat jako objekty ekvivalentních tříd. V případě typu `int` je to třída `Integer`. [2]

Kolekce obsažené v balíku `java.util` obsahují několik rozhraní a tříd, ty nejdůležitější jsou zobrazeny na následujícím obrázku:



Obrázek 2 .2: Vybrané rozhraní a třídy kontejnerů

Prvním rozhraním je `Set` (česky množina), které smí obsahovat každý prvek pouze jednou a obecně nemá žádnou určenou polohu v množině (to neplatí u setřízených množin). Dalším rozhraním je `List` (seznam). Zde je ukládání provedeno na základě indexu. Následuje rozhraní `Queue`, které umožňuje implementaci fronty, čili paměti typu FIFO. Podobným rozhraním je `Deque`. To reprezentuje oboustrannou frontu, do které můžeme vkládat nebo odstraňovat prvky jak ze začátku, tak z konce fronty. Posledním rozhraním je `Map`, jehož ukládání funguje na principu klíč-hodnota.

2.5 Generické typy

Generické typy se objevily poprvé v Javě verze 5. Pojem generické typy v podstatě znamená parametrizované typy, protože nám umožňují vytvářet parametrizované třídy, rozhraní a metody. Parametrem je daný datový typ, se kterým jednotlivé prvky pracují. [5]

Existuje několik situací a algoritmů, které pracují pořád stejně ať už pracují s jakýmkoliv datovým typem. Např. metoda, která pouze vypisuje dané prvky z kolekce nebo třída, která reprezentuje zásobník nebo jinou datovou strukturu. Zásobník bude ve své podstatě fungovat pořád stejně, ale pouze při jeho vytváření mu typovým parametrem předáme informaci jaká data bude ukládat.

Od Javy verze 5 byla také přidána ke všem nadefinovaným kolekcím jejich generická verze. Výhodou těchto generických kolekcí je, že již neukládají objekty typu `Object`, ale objekty typu zadaného typovým parametrem. To pro nás znamená, že např. při procházení celé kolekce pomocí iterátoru již nemusíme jednotlivé prvky přetypovávat, ale vrací se nám objekty přímo daného datového typu. A také nám není umožněno omylem uložit objekt jiného datového typu, než jsme určili v typovém parametru.

Mezi omezení generických typů patří, že nemůžeme jako typový parametr použít primitivní typy, proto podobně jako při ukládání do kolekcí, musíme použít jejich objektové verze - např. pro `int` třídu `Integer`, pro `double` třídu `Double` apod. Také nelze v generické třídě vytvořit pole, které by bylo daného generického typu.

Při definovaní typových parametrů můžeme nastavit různá omezení ve smyslu, že třída uvedená v parametru musí dědit z nějaké konkrétní třídy nebo implementovat dané rozhraní. Těmto omezením se poté říká ohraničení a definují se pomocí klíčového slova `extends`. Podobně také můžeme definovat ohraničení z opačné strany pomocí slova `super` - tím potom řekneme, že třída předaná parametrem musí být předkem dané třídy.

Poslední novinkou u generických typů jsou zástupné znaky nebo také žolíci (v angličtině wildcards). Jako zástupný znak se používá znak ?. Žolík se používá jako neurčitý prvek v generických kolekcích. Pokud chceme například v metodě přijímat parametr datového typu List<> s různými typovými parametry, tak musíme parametr nadefinovat jako `List<?>`, nikoliv jako `List<Object>`, jak by se mohlo zdát. Příkladem použití může být třeba situace, kdy máme několik kolekcí různých tvarů, které všechny obsahují metodu `draw()`. Chceme vytvořit metodu, která projde celou kolekci a zavolá onu metodu. Abychom mohli předat kolekce s různým typovým parametrem, nadefinujeme metodě parametr typu `List<?>`, popřípadě použijeme vhodné ohraničení.

2.6 Výjimky

"Výjimka je chyba, k níž dojde za běhu programu." [5] Výjimky se dají v Javě zpracovat a to znamená, že vyvolání výjimky, tedy chyby, nemusí skončit pádem programu, ale tím, že proběhne určitý blok kódu a program může nějakým způsobem pokračovat, např. návratem do menu.

Základní dělení výjimek je na výjimky kontrolované (checked) a nekontrolované (unchecked). Kontrolované výjimky jsme povinní nějakým způsobem zpracovat. Jednotlivé výjimky jsou třídy, které mají společného předka, a to třídu `Throwable`. Z této třídy dědí `Error` a `Exception`. Výjimky typu `Error` jsou chyby, které vznikají v JVM a nejsou pod kontrolou programátora, spadají tedy do kategorie nekontrolovaných. Výjimky typu `Exception` vznikají během činnosti programu a je umožněno jejich zpracování. Speciální podtřídou `Exception` je třída `RuntimeException`, neboli běhové výjimky. Sem řadíme výjimky, které většinou aplikace nemůže očekávat a předcházet jim. Často vznikají logickými chybami programátora nebo špatným použitím API. Tyto výjimky patří

do kategorie nekontrolovaných, takže programátor není nucen je zpracovat, ale v případě potřeby mu to je umožněno. [9]

Programátor může používat buď předdefinované výjimky nebo si vytvořit vlastní. Vlastní výjimky se vytváří tím, že se vytvoří nová třída dědící z `Exception`.

Odchytávání výjimek probíhá v bloku *try-catch*, případě *try-catch-finally*. Kritická část kódu, ve které je výjimka vyvolávána se označí do bloku `try` a v bloku `catch` se definuje samotné zpracování. Po vyvolání výjimky se okamžitě skočí do odpovídajícího bloku `catch`. Blok `finally` značí blok kódu, který se provede vždy, ať už je výjimka vyvolána, či nikoliv. V případě, že programátor nechce odchytávat výjimku v dané metodě třídy, tak může metodu označit klauzulí `throws` a poté bude muset výjimku ošetřit na místě volání dané metody.

V Javě verze 7 byl představen nový blok `try` s prostředky (try-with-resources), který umožňuje definovat v závorce za slovem `try` uzavíratelné prostředky, např. pro práci se soubory. Tyto prostředky musejí implementovat rozhraní `Closeable`, které obsahuje metodu `close()`. Výhodou tohoto bloku `try` je, že se nám automaticky zavolá metoda `close()` a nemusíme ji explicitně volat v bloku `finally`.

2.7 Datové proudy

Pro vstup a výstup se v Javě používají datové proudy. "Proud je abstrakce, která produkuje nebo konzumuje informace." [5] Proud (stream) si můžeme představit jako kanál, kterým proudí informace. V případě, že aplikace chce číst data, otvírá se vstupní proud a při zápisu se vytváří výstupní proud. [3] Proudy můžou mít různé zdroje a cíle - např. soubory na pevném disku, paměť počítače, ostatní programy nebo třeba sockety síťových aplikací.

Základní rozdělení je na proudy znaků a proudy bajtů. Každý z těchto dvou typů proudů má vlastní hierarchii tříd, které se nalézají v balíčku `java.io`. Pro bajtové proudy existují dvě základní abstraktní třídy `InputStream` a `OutputStream` a pro znakové proudy jsou to abstraktní třídy `Reader` a `Writer`. Od těchto abstraktních tříd je odvozeno několik konkrétních podtříd, které se dělí na dva typy. Prvním typem jsou třídy pro fyzický přenos dat na určité zařízení nebo z něj. Druhý typ jsou třídy pomocné, které slouží pro určité zpracování dat a jsou označovány jako filtry. Do filtrovacích tříd patří např. třídy pro bufferování, formátování, práci s danými datovými typy, práci s objekty nebo třeba pro kompresi dat. [3] Jednotlivé proudy se většinou při vytváření předávají v konstruktorech a tím se vlastnosti řetězí. Třídní diagram vybraných tříd pro práci s proudy viz. Příloha A.

Nejčastěji s proudy pracujeme při výpisu na konzoli nebo při čtení vstupních dat z klávesnice. Dalším typickým použitím je ukládání do souborů a načítání z nich. Proudy se také používají při komunikaci po síti u typických aplikací typu Server-Klient. Při jakékoliv práci s proudy je nutné otevřené proudy po skončení práce korektně uzavřít metodou `close()`. Toto uzavření se většinou provádí v bloku `finally` anebo s využitím bloku `try` se zdroji, kde se metoda `close()` volá automaticky.

2.8 Vlákna

Vlákno (thread) je každá paralelně běžící část programu, které definuje samostatnou cestu provádění. Cílem paralelního zpracování je využít čas nečinnosti procesoru a tím zrychlit celou aplikaci. V jednojádrových systémech každé vlákno obdrží určitý procesorový čas a při vykonávání se vlákna střídají, takže ve skutečnosti paralelně neběží. U vícejádrových systémů je skutečný paralelní běh vláken možný. [5]

Pro práci s vlákny se v Javě používá třída `Thread` a rozhraní `Runnable` z balíčku `java.lang`. Část programu, která má běžet paralelně je vždy naimplementována v metodě `run()`. Pro vytvoření vláken máme několik možností. První je zdědit z třídy `Thread` a v této zděděné třídě přepsat metodu `run()`. To nám ale zabrání dědit z další třídy, a proto druhou možností je, aby třída implementovala rozhraní `Runnable`, které také obsahuje metodu `run()`. Poté pro spuštění je potřeba vytvořit instanci třídy `Thread` a v konstruktoru jí předat objekt implementující `Runnable`. Vlákno se spouští zavoláním metody `start()`. Konec vlákna nastane po skončení metody `run()` nebo zavoláním metody `interrupt()`.

Při vícevláknovém vývoji může nastat situace, kdy potřebujeme, aby k dané části přistoupilo vždy jen jedno vlákno. Typickým příkladem může být práce se sdíleným úložištěm (např. soubor). Pro synchronizace vláken se používá klíčové slovo `synchronized`. Máme dvě možnosti jak jej použít. Buď můžeme označit celou metodu jako `synchronized` anebo označíme blok kódu s kritickou sekcí pomocí klíčového slova `synchronized`.

Další potřebnou věcí pro vývoj s vlákny je komunikace pomocí metod `wait()`, `notify()` a `notifyAll()`. Pomocí těchto metod můžeme pozastavit provádění všech vláken a následně jim dát zase vědět, že můžou pokračovat. Typickým příkladem je program Producent-Konzument, kdy některé vlákna čtou data a jiná je generují. V případě, že vlákno nemá co číst, se zavolá `wait()` a až se vygenerují data, se zavolá `notify()` nebo `notifyAll()`.

3 Objektové rysy jazyka

3.1 Zadání 1.1 - přetěžování konstruktorů

Specifikace

Vytvořte třídu `Person` s privátními atributy `firstName`, `lastName` a `identificationNumber`. Pro každý atribut vytvořte odpovídající get a set metody. Vytvořte čtyři konstruktory přijímající různé parametry. V každém konstruktoru volejte konstruktor se třemi parametry, nenastavené parametry nastavte do výchozích hodnot (*null*, 0). V každém konstruktoru vypište do konzole, o který konstruktor se jedná. Ve třídě `Person` také vytvořte metodu `print()`, která vypíše všechny hodnoty dané instance do konzole.

Doplňte třídu `Person` o privátní statický atribut `numberOfInstances` a odpovídající get metodu. Při vytvoření instance v konstruktoru tuto hodnotu inkrementujte.

Testování

Vytvořte třídu `Main` se spouštěnou metodou `public static void main(String[] args)`, ve které vytvořte čtyři instance třídy Person. Pokaždé využijte jiný konstruktor, poté na každou instanci zavolejte metodu `print()`. Dále si nechte vypsat do konzole hodnoty atributu `numberOfInstances` pro každou instanci a dané výpisy porovnejte.

Omezení: žádné.

Obtížnost: 1 - začátečnická.

Zahrnuté oblasti: konstruktory, get/set metody, statické a nestatické atributy.

3.2 Zadání 1.2 - dědičnost

Specifikace

Vytvořte abstraktní třídu `Shapes` s privátním atributem `color`, parametrickým i bezparametrickým konstruktorem a abstraktními metodami `print()`, `getPerimeter()` a `getArea()`. Vytvořte třídu `Rectangle`, která bude dědit z třídy `Shape` a bude obsahovat chráněné atributy `sideA` a `sideB`. V parametrickém konstruktoru nastavte všechny atributy a zavolejte konstruktor předka, ve kterém předáte parametr `color`. Zděděné abstraktní metody doplňte o implementaci, tak aby správně spočítaly obvod a obsah. V metodě `print()` vypište, že se jedná o obdélník a všechny jeho parametry. Z třídy `Rectangle` bude dále dědit třída `Square`, u které zajistěte, aby z ní nešlo již dále dědit. V jejím konstruktoru zavolejte konstruktor předka, kde předáte dvakrát hodnotu strany A. Přepište pouze metodu `print()`. Všimněte si, ke kterým atributům předků můžete přistupovat přímo a ke kterým pouze přes get metody. Poslední zděděnou třídou bude `Circle`, která bude dědit ze třídy `Shapes`. Třída bude obsahovat privátní atribut `radius` a zděděné funkce, které doplňte o správnou implementaci.

Testování

Vytvořte třídu `Main` se spouštěnou metodou `main()`, ve které vytvořte instance jednotlivých tříd a poté na ně zavolejte metodu `print()`. Následně vytvořte nové instance a uložte je do proměnných datového typu `Shapes` a opět zavolejte metodu `print()`. Vypsané výsledky porovnejte.

Omezení: poloměr a strany jsou celočíselné atributy; při vytváření nových tvarů je nutno zadat všechny odpovídající atributy.

Obtížnost: 2 - mírně pokročilá.

Zahrnuté oblasti: dědičnost, protected atributy, abstract a final třídy, abstract metody.

3.3 Zadání 1.3 - rozhraní

Specifikace

Vytvořte rozhraní `IAgeCalculator`, obsahující metodu `getAge()` a rozhraní `IPrinter` s metodou `print()`. Napište třídu `Person`, implementující rozhraní `IPrinter`. Třída bude obsahovat privátní atributy `firstName` a `lastName`. Naimplementujte metodu z rozhraní tak, aby vypisovala údaje o dané osobě. Vytvořte druhou třídu `PersonWithDate` obdobně jako třídu `Person`. Třída bude navíc implementovat rozhraní `IAgeCalculator`, a proto přidejte celočíselný atribut `yearOfBirth`. V metodě `getAge()` vypočítejte aktuální věk osoby.

Ukázka získání aktuálního roku:

```
DateFormat dateFormat = new SimpleDateFormat("yyyy");
Data date = new Date();
int currentYear = Integer.parseInt(dateFormat.format(date));
```

Testování

Ve třídě `Main` a spouštěné metodě `main()` vše otestujte vytvořením instancí obou tříd a zavoláním příslušných metod. Vyzkoušejte také uložení instancí do proměnných datového typu jednotlivých rozhraní.

Omezení: věk se počítá pouze na základě roku narození; při vytváření osob je nutno zadat všechny odpovídající atributy.

Obtížnost: 1 - začátečnická.

Zahrnuté oblasti: rozhraní, práce s aktuálním datem.

3.4 Zadání 1.4 - balíčky

Specifikace

Vytvořte si dva balíčky `persons1` a `persons2`. Do každého balíčku vytvořte třídu `Student` s jinou implementací - v jedné použijte atributy `name` a `mail` a v druhé `id` a `name`.

V každé třídě vytvořte metodu, pomocí které budete moct vypsat daného studenta do konzole. Dále vytvořte balíček `shapes1`, do kterého naprogramujte třídy `Shapes`, `Circle` a `Rectangle` podobně jako v zadání 1.2 (stačí část implementace). Nakonec vytvořte balíček `shapes2`, ve kterém vytvoříte třídu `Shapes` a další balíček `concreteShapes`, který bude obsahovat třídy `Circle` a `Rectangle`.

Testování

Vytvořte balíček `main` a v něm třídy `Main1`, `Main2`, `Main3` a `Main4` vždy se spouštěcí metodou `main()`. V první třídě si vyzkoušejte vytvořit instance stejně pojmenovaných tříd `Student` z různých balíčků. Ve třídě `Main2` naimportujte třídu `Circle` z balíčku `shapes1` a vyzkoušejte rozdíl při vytváření instancí třídy `Circle` a `Rectangle`. V další třídě `Main3` naimportujte celý balíček `concreteShapes` z balíčku `shapes2` a vyzkoušejte, ke kterým třídám máte přímý přístup. Nakonec ve třídě `Main4` naimportujte celý balíček `shapes2` a vyzkoušejte zda se dostanete i ke třídám z vnořeného balíčku.

Omezení: žádné.

Obtížnost: 1 - začátečnická.

Zahrnuté oblasti: balíčky, import, dědičnost.

3.5 Zadání 1.5 - překrývání metod třídy Object

Specifikace

Vytvořte třídu `Fraction` s celočíselnými atributy `numerator` a `denominator`. Hodnoty nastavujte v konstruktoru a také vytvořte odpovídající set a get metody. Třída bude dále obsahovat metodu `gcd()`, vracející celočíselný největší společný dělitel zlomku a metodu `reduction()`, která daný zlomek zkrátí. Ve třídě `Fraction` dále překryjte metody třídy `Object`, konkrétně `toString()`, která bude vracet popis zlomku, a `equals(Object obj)`, která vrátí *true* pokud je v parametru stejný zlomek. Nezapomeňte, že přijímáte parametr typu `Object`, a proto je potřeba ověřit, zda se jedná o instanci třídy `Fraction` a správně jej přetypovat. Pokud se překryje metoda `equals()`, tak by se měla správně překrýt i metoda `hashCode()`, která vrací celočíselný hash dané instance. Implementaci proveďte následovně: vytvořte si nenulovou celočíselnou proměnnou a poté k ní postupně přičtěte vždy 37-násobek aktuální hodnoty a hodnotu privátního atributu. V našem případě budete přičítat dvakrát, jednou hodnotu numeratoru a jednou denominatoru. Nakonec vypočítanou hodnotu vraťte.

Dále vytvořte třídu `FractionOperation`, ve které nadefinujte čtyři statické metody přijímající dva parametry typu `Fraction` pro sčítání, odečítání, násobení a dělení.

Veškeré třídy a metody okomentujte podle pravidel Javadoc.

Testování

V metodě `main()` vytvořte dva stejné zlomky a jeden odlišný. Zkuste je navzájem porovnat. Zkuste si zlomky vypsat do konzole, všimněte si, že nemusíte explicitně volat metodu `toString()`.

Nad dvěma zlomky proveďte všechny operace z třídy `FractionOperation` a výsledky zkontrolujte.

Omezení: operace se zlomky probíhají vždy jen nad dvěma zlomky; při vytváření zlomku je nutno zadat jeho hodnotu.

Obtížnost: 2 - mírně pokročilá.

Zahrnuté oblasti: Object, operátor instanceof, statické metody, javadoc.

3.6 Zadání 1.6 - vnořené třídy, výčtový typ

Specifikace

Jako první si vytvořte výčtový typ `Department` s několika odděleními (např. accountant, IT, financial, social atd.). Dále si vytvořte třídu `Employee` s atributy `department` datového typu `Department`, `salary` s datovým typem `Salary` a `name` typu `String`. Ve třídě `Employee` nadefinujte vnořenou třídu `Salary` s atributy `amount` a `currency`. Vytvořte parametrický a bezparametrický konstruktor, get a set metody a metodu `getPayment(int hours)`, ve které vrátíte vypočítanou výplatu za daný počet hodin. Ve vnořené třídě si vyzkoušejte, že se přímo dostanete i k privátním atributům vnější třídy. Ve vnější třídě `Employee`, nadefinujte odpovídající get a set metody, překryjte metodu `toString()` a vytvořte metodu `raise(int amount)`, která zvedne plat o předané množství. Opět si vyzkoušejte, že se dostanete ke všem atributům vnořené třídy.

Nakonec si vytvořte rozhraní `ICalculator` s metodami přijímajícími dva argumenty typu double pro matematické operace sčítání, odečítání, násobení a dělení. V metodě `main()` třídy `Main` naimplementujte toto rozhraní jako anonymní třídu.

Testování

V metodě `main()` si vytvořte zaměstnance. Přiřaďte mu jméno, plat, měnu a oddělení. Zaměstnance vypište, vypište také jeho výplatu za určitý počet hodin, poté mu plat zvyšte a opět vypište výplatu. Zkontrolujte, zda vše funguje správně. Jako poslední vyzkoušejte kalkulačku zde naimplementovanou jako anonymní třídu.

Omezení: metody kalkulačky přijímají vždy jen dva atributy; při vytváření zaměstnance je nutno zadat jméno.

Obtížnost: 2 - mírně pokročilá.

Zahrnuté oblasti: výčtový typ, vnořené třídy, anonymní třídy, rozhraní.

3.7 Zadání 1.7 - stromová struktura

Specifikace

Vytvořte stromovou strukturu HTML dokumentu pomocí dědičnosti. Vytvořte abstraktní třídu `Node` s atributem `content` typu `String` a kolekcí potomků children. Protože kolekce jsou

předmětem další kategorie, tak použijte pro uložení a přidávání jednoduchý `Vector`. Ukázka práce s Vectorem viz. Příloha B.

Ve třídě vytvořte metodu pro přidání `Node` do kolekce, metodu `renderContent()`, která vypíše do konzole `content`, abstraktní metody `renderBeforeContent()` a `renderAfterContent()` a metodu `render()`. Metoda `render()` zavolá metodu `renderBeforeContent()`, poté `renderContent()`, pak projde všechny prvky kolekce a zavolá na ně metodu `render()` a nakonec zavolá metodu `renderAfterContent()`.

Dále si vytvořte třídy, které budou dědit z třídy `Node`. V těchto třídách naimplementujte metody `renderBeforeContent()` a `renderAfterContent()`, ve kterých budete vypisovat počáteční a koncové tagové značky. Vytvořte třídy např. `NodeHTML`, `NodeBODY`, `NodeLI` a `NodeUL`.

Testování

V metodě `main()` si vytvořte jednotlivé instance a správně je zanořte do ostatních - do *html* vložte *body*, do *body* vložte *ul*, do *ul* vložte několik *li* atd. Nakonec zavolejte na kořen (v našem případě by to měla být instance třídy `NodeHTML`) metodu `render()` a zkontrolujte správnost výpisu.

Omezení: žádné.

Obtížnost: 2 - mírně pokročilá.

Zahrnuté oblasti: dědičnost, abstraktní třídy a metody, Vector.

3.8 Zadání 1.8 - události

Specifikace

Vytvořte třídu `Counter` s metodou `count()`, která bude počítat čísla od 1 až do zadaného limitu, předaného v konstruktoru. Pokud bude aktuální číslo sudé nebo prvočíslo, vyvolá se událost příslušným posluchačům. Posluchači budou dva, jeden pro sudé čísla a druhý pro prvočísla. Každý musí implementovat rozhraní `ICounterListener`, které rozšiřuje rozhraní `EventListener`. V rozhraní vytvořte metodu `print(CounterEvent ev)`, kterou pak v posluchačích naimplementujte. Třída `CounterEvent` musí dědit ze třídy `EventObject` a v konstruktoru musí volat konstruktor předka s parametrem typu `Object` a hodnotou objektu, který událost vyvolal. Třída `CounterEvent` bude obsahovat atribut `number` a příslušnou get metodu.

Ve třídě `Counter` nadefinujte dvě kolekce pro dva typy posluchačů (použijte podobně jako v minulém zadání kolekci `Vector<ICounterListener>`) a metody pro přidání a odebrání z kolekce. Budete potřebovat také metody `firePrimeNumberEvent(int n)` a `fireEvenNumberEvent(int n)`, které vytvoří nový objekt typu `CounterEvent` a projdou celou kolekci daných posluchačů a zavolají na ně metodu `print()`. V metodě `count()` tedy počítejte čísla a ověřujte, jestli se jedná o prvočísla nebo sudá čísla a v případě, že ano, volejte odpovídající metody `firePrimeNumberEvent()` nebo `fireEvenNumberEvent()`.

Testování

V metodě `main()` si vytvořte instanci třídy `Counter` a dané posluchače, které zaregistrujte do objektu typu `Counter`. Zavolejte metodu `count()` a sledujte výpis v konzoli a ověřte, zda se události správně volají.

Omezení: nutno zaregistrovat posluchače, aby bylo možno události odchytávat; počítání čísel probíhá vždy od 1 do zadaného limitu - nelze přímo nastavit rozmezí.

Obtížnost: 3 - pokročilá.

Zahrnuté oblasti: události, rozhraní, Vector.

3.9 Zadání 1.9 - návrhový vzor singleton

Specifikace

Vytvořte třídu `PrinterManager`, která bude obsahovat několik tiskáren. Vytvořte tiskárny `PrinterPDF`, `PrinterColor`, `PrinterBlackAndWhite` a `PrinterPhoto`. Jednotlivé tiskárny vytvořte jako potomky abstraktní třídy `Printer`, která obsahuje abstraktní metodu `print()`, v té vždy vypište, co tisknete.

Zajistěte, aby existovala vždy jen jedna instance třídy `PrinterManager`. Toho docílíte tak, že nadefinujete konstruktor jako privátní, vytvoříte si statickou proměnnou typu `PrinterManager`, která bude udržovat právě tu jednu instanci, a statickou metodu `getPrinterManager()`, ve které ověříte zda již existuje instance nebo ne, pokud ano, tak ji vrátíte a pokud ne, tak ji vytvoříte.

Testování

V metodě `main()` si zkuste vytvořit klasicky instanci třídy `PrinterManager`. Poté získejte dvě instance pomocí statické metody `getPrinterManager()`. Ve třídě `PrinterManager` si můžete doplnit statickou proměnnou, která bude počítat jednotlivé instance a v metodě `main()` se pomocí ní přesvědčit, že je vytvořena opravdu jen jedna instance.

Omezení: při vytváření tiskárny je nutno zadat název.

Obtížnost: 2 - mírně pokročilá.

Zahrnuté oblasti: dědičnost, návrhový vzor singleton.

3.10 Zadání 1.10 - návrhový vzor mediator

Specifikace

Vaším úkolem je naprogramovat chatovací místnost a uživatele, chatovací místnost bude fungovat jako mediator. Třídní diagram vzoru mediator naleznete v Příloze C. Vytvořte si rozhraní `IChatroom` se dvěma metodami `send(String from, String to, String message)` a metodu `register(AbstractUser user)` pro přidání uživatele do chatovací místnosti. Vytvořte třídu `Chatroom` implementující rozhraní `IChatroom`, která bude obsahovat uživatele v kolekci `Vector` (viz. zadání 1.7). Pro uživatele si vytvořte abstraktní třídu `AbstractUser`

s chráněnými atributy `name` a `chatroom`. Naprogramujte get a set metody a navíc metody `send(String to, String message)` a `receive(String from, String message)`. V metodě `send()` pouze zavolejte metodu `send()` na objekt `chatroom`. Metoda `receive()` bude pouze vypisovat do konzole, kdo komu poslal jakou zprávu. Dále si vytvořte konkrétní třídu pro uživatele dědící z `AbstractUser`, například třídu `UserWithSignature` a implementaci doplňte o podpis.

Nyní se vraťte k třídě `Chatroom`, kde doimplementujte metody `send()` a `register()`. V metodě `register()` pouze přidejte uživatele do kolekce `Vector`. V metodě `send()`, projdete všechny uživatele, naleznete toho, komu se má zpráva poslat a zavoláte na něm metodu `receive()`.

Testování

V metodě `main()` si vytvořte chatovací místnost, několik uživatelů, ty poté zaregistrujte do chatovací místnosti a pošlete několik zpráv ze všech uživatelů. V konzoli si zkontrolujte výpisy, že vám vše funguje.

Omezení: před posíláním zpráv je nutno zaregistrovat uživatele; při vytváření uživatele je nutno zadat jméno.

Obtížnost: 3 - pokročilá.

Zahrnuté oblasti: návrhový vzor mediator, Vector, dědičnost, rozhraní.

4 Kontejnery

4.1 Zadání 2.1 - zásobník

Specifikace

Naprogramujte zásobník implementovaný pomocí pole s pevnou délkou. Vytvořte si rozhraní `IStack` s charakteristickými metodami pro zásobník - `push(int number)`, `pop()`, `top()`, `isFull()`, `isEmpty()` a `clear()`. Vytvořte třídu `Stack`, implementující rozhraní `IStack`. Třída bude obsahovat pole datového typu `int`. V konstruktoru vytvořte pole o velikosti předané parametrem. Nedovolte podtečení nebo přetečení zásobníku - při vkládání a odebírání ověřte stav zásobníku pomocí metod `isFull()` a `isEmpty()`.

Testování

V metodě `main()` vytvořte instanci zásobníku o velikosti např. tři. Ověřte, že metody `isFull()` a `isEmpty()` fungují správně. Vložte do zásobníku hodnoty a ověřte, že nedojde k přetečení. Znovu zavolejte metody `isFull()` a `isEmpty()`. Dále vyzkoušejte metodu `top()`, že opravdu neodebere prvek ze zásobníku, ale pouze ho vypíše. Nakonec odeberte všechny prvky ze zásobníku a vyzkoušejte odebrání i z prázdného zásobníku.

Omezení: zásobník ukládá pouze prvky datového typu `int` a má omezenou velikost.

Obtížnost: 1 - začátečnická.

Zahrnuté oblasti: zásobník, pole, rozhraní.

4.2 Zadání 2.2 - fronta

Specifikace

Naprogramujte frontu, implementovanou podobně jako zásobník v zadání 2.1. Napřed si vytvořte rozhraní `IQueue` s metodami `enqueue(int number)`, `dequeue()`, `isFull()`, `isEmpty()` a `clear()`. Třída `Queue` bude toto rozhraní implementovat. V konstruktoru vytvořte pole datového typu `int` o zadané velikost. Pro vkládání a odebírání z fronty budete potřebovat dva ukazatele `head` a `tail`. V metodách `enqueue()` a `dequeue()` zajistěte, aby nedošlo k přetečení nebo podtečení fronty. U takto implementované fronty o velikosti třeba deset dojde k problému, že např. po pěti vložených a odebraných prvcích zůstane ve frontě pouze pět volných míst místo deseti, proto tenhle problém vyřešte metodou `reorder()`, která přeskládá frontu opět od nulté pozice pole. Metodu volejte v případě, že chcete uložit prvek, ale není kam a zároveň počet uložených prvků se nerovná velikosti fronty.

Testování

Frontu otestujte obdobně jako zásobník v zadání 2.1. Vytvořte si frontu o dané velikosti, zavolejte metody `isFull()` a `isEmpty()`. Vložte několik prvků a ověřte, že fronta nepřeteče. Poté opět zavolejte metody `isFull()` a `isEmpty()`. Prvky následně vyjměte z fronty a zkontrolujte, že

se odebírají ve správném pořadí. Zkuste vložit další prvky do fronty, pokud se vám ji podaří zcela naplnit, tak to znamená, že jste správně vytvořili metodu `reorder()`.

Omezení: fronta ukládá pouze prvky datového typu `int` a má omezenou velikost.

Obtížnost: 2 - mírně pokročilá.

Zahrnuté oblasti: fronta, pole, rozhraní.

4.3 Zadání 2.3 - ukládání primitivních typů do kolekcí

Specifikace

Vytvořte třídu `Student` s privátním atributem `name` a privátními kolekcemi `mathGrades`, `electronicGrades`, `historyGrades` a `grantedScolarships` typu `Vector`. Třída bude obsahovat metodu `addGrade(String subject, int grade)`, která na základě předmětu přidá známku do správné kolekce. Metoda `grantScholarship(double amount)` přidá do dané kolekce stipendium. Další metodou bude `getAverageGrade(String subject)`, která vrátí průměr známek z daného předmětu. Dále zde bude metoda `getTotalScholarships()`, která vrátí celkovou hodnotu všech přiznaných stipendií. Nakonec vytvořte metody `getHighestScholarship()` a `getLowestScholarship()`, které budou vracet odpovídající hodnoty.

Testování

V metodě `main()` si vytvořte studenta a přidejte mu ke každému předmětu několik známek. Nechte si vypsat jednotlivé průměry známek a ověřte, že daný výpis je správný. Přidělte také studentovi nějaká stipendia a vypište si nejnižší, nejvyšší a celkové.

Omezení: při vytváření studenta je nutno zadat jméno.

Obtížnost: 2 - mírně pokročilá.

Zahrnuté oblasti: Vector.

4.4 Zadání 2.4 - LinkedList a iterátor

Specifikace

Vytvořte abstraktní třídu `Employee` s privátními atributy `name` a `salary` a příslušnými get metodami. Tyto hodnoty nastavujte v konstruktoru. Vytvořte zde také abstraktní metodu `work()`. Z třídy `Employee` budou dědit `Manager`, `Accountant` a `Technician`. V každé této třídě v konstruktoru zavolejte konstruktor předka s danými parametry a naimplementujte metodu `work()`, která bude pouze vypisovat, že daný pracovník pracuje.

Dále si vytvořte třídu `Company` s privátním atributem `employees` datového typu `List`. V konstruktoru uložte do tohoto atributu nový `LinkedList`. Vytvořte metody `addEmployee(Employee e)`, `payment()`, `work()` a `listAllEmployees()`. V metodě `addEmployee()` pouze přidejte zaměstnance do kolekce. Metoda `listAllEmployees()` vypíše

na konzoli všechny zaměstnance kolekce, pro výpis použijte `Iterator` a cyklus `for`. V metodě `work()` zavolejte na všechny zaměstnance metodu `work()`, pro průchod použijte `Iterator` a cyklus `while`. Metoda `payment()` projde všechny zaměstnance a vypíše jejich plat, zde použijte pro průchod cyklus foreach.

Testování

V metodě `main()` vytvořte jednu firmu a několik různých zaměstnanců, které do firmy přidáte. Nakonec zavolejte na firmu metody `listAllEmployees()`, `work()` a `payment()` a jednotlivé výpisy zkontrolujte.

Omezení: při vytváření zaměstnance je nutno zadat jméno a plat.

Obtížnost: 2 - mírně pokročilá.

Zahrnuté oblasti: LinkedList, Iterator, foreach, dědičnost.

4.5 Zadání 2.5 - ArrayList

Specifikace

Vytvořte třídu `Account` s atributy `number` a `balance`. Hodnoty nastavujte v konstruktoru a umožněte i jejich nastavení a získání pomocí set a get metod. Dále zde vytvořte metody pro vložení peněz a pro vybrání peněz z účtu. Přepište také metodu `toString()`.

Vytvořte druhou třídu `Bank`, která bude spravovat jednotlivé účty. Vytvořte si v ní tedy kolekci účtů za pomoci ArrayListu. Naprogramujte metodu `addAccount()`, která vytvoří nový účet, automaticky mu přidělí číslo, vloží jej do kolekce a vypíše informaci o novém účtu. Další metody obsažené v třídě budou zajišťovat odstranění účtu z banky, vypsání aktuálního zůstatku na účtu, výběr z účtu a vložení na účet. Protože rozhraní `List` umožňuje přistupovat k prvkům na základě indexu a implementace pomocí třídy `ArrayList` je k tomu vhodná, proto vytvořte vždy dvě verze těchto metod. Jedna verze bude přistupovat k účtu na základě pozice v seznamu a druhá verze na základě čísla účtu. Nakonec ještě vytvořte metodu `listAllAccounts()`, která projde všechny účty a vypíše je.

Testování

Vytvořte si instanci třídy `Bank`, vygenerujte několik účtů. Zkuste si odstranit účty na daných pozicích, přidat další účty, vložit a vybrat peníze z účtů. Vždy si vše kontrolujte výpisem všech účtů. Všimněte si, jak se změní uspořádání účtů po odstranění na nějaké pozici.

Omezení: při vytváření účtu je nutno zadat číslo.

Obtížnost: 2 - mírně pokročilá.

Zahrnuté oblasti: ArrayList, procházení kolekce, dědičnost.

4.6 Zadání 2.6 - HashSet

Specifikace

Naprogramujte malou pizzerii přijímající objednávky. Vytvořte třídu `PizzaOrder`, která bude obsahovat `orderNumber` typu `int`, `customer` typu `String` a `List` objednaných pizz. Napište get a set metody a implementaci metody `toString()` pro vypsání. Dále si vytvořte třídu `Pizzeria` se třemi atributy typu `Set`, budou to internetové objednávky, přijaté objednávky a dokončené objednávky. V konstruktoru vytvořte do těchto atributů instance třídy `HashSet`. Pro dokončené objednávky nastavte velikost na 300. Vytvořte metody pro výpis jednotlivých množin do konzole, metodu pro přidání internetové objednávky, metodu pro potvrzení objednávky, která ji přesune z množiny internetových do množiny přijatých objednávek, a nakonec metodu pro dokončení objednávky, která ji přesune do množiny dokončených objednávek. Metody, které přesouvají jednotlivé objednávky, budou přijímat jako parametr číslo objednávky.

Testování

Vytvořte si instanci `Pizzerie` a několik objednávek s různým počtem pizz, které přidáte do pizzerie. Nechte si vypsat všechny objednávky a všimněte si, jak jsou v množině seřazeny. Vytvořte novou objednávku a některé objednávky přijměte. Znovu si vypište objednávky, zkontrolujte, že se objednávky opravdu přesunuly a také v jakém pořadí se uložily. Nakonec ještě nějakou objednávku ukončete a znovu vše vypište a zkontrolujte.

Omezení: při vytváření objednávky je nutno zadat číslo a zákazníka; přesouvání objednávky na základě čísla objednávky.

Obtížnost: 3 - pokročilá.

Zahrnuté oblasti: HashSet, procházení kolekce.

4.7 Zadání 2.7 - TreeSet

Specifikace

Vaším úkolem je naprogramovat tenisový žebříček. Vytvořte třídu `TennisPlayer` implementující rozhraní `Comparable`. Třída bude mít atributy `rank`, `name` a `nationality`, vytvořte také get a set metody. Dále překryjte metodu `toString()`, abyste mohli následně hráče lehce vypsat. Nakonec musíte naimplementovat metodu `compareTo(Object o)`, která bude porovnávat podle atributu `rank`. Implementaci vytvořte tak, že pokud je konkrétní objekt menší než objekt přijatý přes parametr, vraťte hodnotu -1, pokud jsou si rovny, tak hodnotu 0 a pokud je větší, tak hodnotu 1.

Nyní si vytvořte třídu `RankingWithTreeset`, která bude mít jeden atribut typu `SortedSet`, do kterého uložíte instanci třídy `TreeSet`. Vytvořte metody pro přidání hráče, změnu umístění hráče v žebříčku a metodu pro výpis všech hráčů. Protože se jedná o seřazenou množinu, tak máme k dispozici předdefinované metody pro získání prvního a poslední hráče. Napište si tedy metody, které budou dané hráče za pomocí těchto metod vracet.

Nakonec si vytvořte ještě třídu `RankingOrderedByName`, která bude mít stejnou implementaci jako třída `RankingWithTreeset`, ale s tím rozdílem, že při vytváření nového TreeSetu mu v konstruktoru předáte instanci třídy `Comparator` porovnávající hráče na základě jména. Třídu `Comparator` s metodou `compare(Object o1, Object o2)` naimplementujte jako anonymní třídu.

Testování

Jako první si vytvořte instanci třídy `RankingWithTreeset` a několik hráčů, které do ní postupně vložíte. Hráče vkládejte tak, aby jejich `rank` nešel postupně. Hráče si vypište a zkontrolujte, že se opravdu seřadili. Nyní změňte pořadí hráčů a znovu je vypište. Všimněte si, že se pořadí hráčů v množině nezměnilo. Vložte dalšího hráče tak, aby jeho `rank` nebyl nejnižší a poté vypište celý žebříček. Z výpisu můžete poznat, že se hráči seřazují pouze v době vložení do stromu a neexistuje žádná hotová metoda pro seřazení.

Nyní vyzkoušejte druhou třídu `RankingOrderedByName`. Proveďte stejné úkony jako v předchozím příkladě. Měli byste ověřit, že se hráči správně seřadili podle jména a při změně atributu `rank` se jejich pořadí ve stromu pochopitelně nemění.

Omezení: při vytváření hráče je nutno zadat umístění, jméno a národnost; hráči se seřazují pouze v době přidání do kolekce.

Obtížnost: 3 - pokročilá.

Zahrnuté oblasti: TreeSet, procházení kolekce, Comparable, Comparator, anonymní třídy.

4.8 Zadání 2.8 - HashMap

Specifikace

Naprogramujte telefonní seznam s využitím kolekce typu `Map`. Vytvořte třídu `Record` s atributy `name`, `cellNumber`, `homeNumber`, `workNumber` a `email`. Naprogramujte odpovídající konstruktor, get a set metody a metodu, pomocí které záznam vypíšete do konzole (můžete využít `toString()`). Dále vytvořte třídu `Phonebook` s atributem `phonebook` typu `Map`, do kterého uložte instanci `HashMap`. Třída bude obsahovat metodu pro přidání záznamu, jako klíč použijte jméno a jako hodnotu objekt typu `Record`. Další metoda bude `getRecordByName(String name)`, která vrátí záznam na základě klíče. Zde ověřte, jestli záznam existuje pomocí hotové metody `containsKey()`. Vytvořte také metodu, která zobrazí všechny klíče z mapy. Nakonec napište ještě metody pro odstranění záznamu a pro zjištění počtu záznamů.

Testování

V metodě `main()` vytvořte nový `Phonebook` a poté několik záznamů, které do něj uložíte. Nechte si vypsat počet záznamů a jednotlivé klíče. Nechte si vypsat nějaký záznam, zkuste také získat záznam, který v kolekci není. Nakonec některý záznam odstraňte a znovu vypište velikost a všechny klíče. Jednotlivé výpisy zkontrolujte a ověřte, že vám vše funguje správně.

Omezení: získání a odstranění záznamu z kolekce pouze na základě jména.

Obtížnost: 2 - mírně pokročilá.

Zahrnuté oblasti: HashMap, procházení kolekce.

4.9 Zadání 2.9 - Collections

Specifikace

Vyzkoušejte si statické metody z třídy `Collections`. Naprogramujte třídu `Employee` s atributy `name` a seznamem `payments`. Vytvořte odpovídající konstruktor, get a set metody. Dále vytvořte metodu přidávající výplatu a metody `getHighestPayment()`, `getLowestPayment()` a `getFrequencyOfPayment(int payment)` využívající metody z `Collections`. Getter pro atribut `payments` bude vracet seznam jako `unmodifiableList`, který neumožňuje vkládání do seznamu. Další metodou bude `getFirstXPayments(int x)`, která bude vracet danou část seznamu - využijte metodu `subList()` volanou na atribut `payments`. Nakonec ještě přidejte metodu `getPaymentsAsArray()` vracející seznam jako pole objektů. K tomu použijte metodu `toArray()`. Třída bude také implementovat rozhraní `Comparable`, aby bylo možno zaměstnance seřazovat. Pro porovnání použijte atribut `name`.

Další třídou bude `PersonalAgenda`, obsahující `List` zaměstnanců. Tento `List` vytvořte v konstruktoru jako *synchronizedList* za pomocí třídy `Collections`. Pro základní práci se seznamem vytvořte metody pro přidání zaměstnance, získání zaměstnance podle jména a vypsání všech zaměstnanců. Nakonec přidejte metody pro třídění seznamu `sortEmployees()`, `reverseEmployees()` a `shuffleEmployees()`, které budou využívat odpovídající metody třídy `Collections`.

Testování

Vytvořte instanci třídy `PersonalAgenda` a několik zaměstnanců s různými výplatami. Zaměstnance postupně přidejte do agendy. Nechte si zaměstnance vypsat, poté je seřadit a opět vypsat. To samé proveďte i s otočením seznamu a zamícháním seznamu. Dále si vyberte jednoho zaměstnance a zjistěte jeho nejvyšší a nejnižší výplatu a také frekvenci určité výplaty. Poté si nechte vrátit jenom určitý počet výplat a ty vypište a také si výplaty vraťte jako pole. Dosavadní výpisy zkontrolujte, že odpovídají zadaným hodnotám. Nakonec si ještě zkuste vrátit seznam výplat pomocí metody `getPayments()` a zkuste přidat novou výplatu. Program by vám to neměl dovolit a vyhodí výjimku `UnsupportedOperationException`.

Omezení: při vytváření zaměstnance je nutno zadat jméno; lze vrátit seznam výplat zaměstnance, ale nelze do něj přímo přidávat hodnoty.

Obtížnost: 2 - mírně pokročilá.

Zahrnuté oblasti: List, Collections, procházení kolekce, Comparable.

4.10 Zadání 2.10 - binární vyhledávací strom

Specifikace

Naprogramujte binární vyhledávací strom. Vytvořte třídu `Node`, která bude reprezentovat uzel stromu. Bude obsahovat atribut `value` datového typu `int` a ukazatel na levý a pravý podstrom. Třída `BinayTree` bude reprezentovat samotný strom a bude obsahovat metody `insert(int value)`, `remove(int value)`, `clear()` a metody, které vypíšou strom třemi druhy průchodu - preorder, inorder a postorder. Pro implementaci můžete využít např. privátní metody, které budete rekurzivně volat.

Testování

V metodě `main()` vytvořený strom otestujte vložením prvků a jednotlivými výpisy. Otestujte také, že správně funguje odstranění zadaného uzlu. Po vložení hodnot 6, 1, 5, 3, 2 a 4 budou výpisy vypadat následovně: preorder - 6, 1, 5, 3, 2, 4; inorder - 1, 2, 3, 4, 5, 6; postorder - 2, 4, 3, 5, 1, 6. Po odstranění hodnoty 3 budou výpisy: preorder - 6, 1, 5, 4, 2; inorder - 1, 2, 4, 5, 6; postorder - 2, 4, 5, 1, 6. A po dalším odstranění hodnoty 4: preorder - 6, 1, 5, 2; inorder - 1, 2, 5, 6; postorder - 2, 5, 1, 6.

Omezení: strom uchovává hodnoty datového typu `int`.

Obtížnost: 5 - těžká.

Zahrnuté oblasti: binární strom, rekurze.

5 Generické typy

5.1 Zadání 3.1 - generický zásobník

Specifikace

V zadání 2.1 bylo vaším úkolem naimplementovat zásobník pomocí pole datového typu `int`. Nyní naimplementujte zásobník se stejnými funkcemi, ale s využitím generických typů. Vytvořte si tedy opět rozhraní, které bude také generické a bude obsahovat metody `push(T item)`, `pop()`, `top()`, `isFull()`, `isEmpty()` a `clear()`. Třída `Stack` bude toto rozhraní implementovat. Vzhledem k tomu, že nelze vytvořit pole s generickým datovým typem, tak pro zásobník použijte generický `ArrayList`. Třída bude také obsahovat atribut `size`, který bude obsahovat informaci o maximální velikosti zásobníku.

Testování

Vytvořte si instanci zásobníku o velikosti 3, který bude uchovávat prvky typu `String`. Zavolejte metody `isEmpty()` a `isFull()`, poté zásobník naplňte. Vyzkoušejte, že metoda `top()` prvek ze zásobníku neodebere, následně všechny prvky odeberte. Ověřte, že nedojde k přetečení nebo podtečení zásobníku.

Vytvořte si další instanci zásobníku s jiným typovým parametrem, např. `Integer`. Vyzkoušejte, že zásobník funguje stejným způsobem. Ověřte také, že vám není dovoleno vložit hodnotu jiného datového typu, než bylo uvedeno při vytvoření instance.

Omezení: zásobník má omezenou velikost zadanou v konstruktoru.

Obtížnost: 2 - mírně pokročilá.

Zahrnuté oblasti: zásobník, generické typy, ArrayList, rozhraní.

5.2 Zadání 3.2 - generická fronta

Specifikace

Podobně jako v zadání 3.1, kde jste vytvářeli generickou verzi zásobníku, nyní vytvořte generickou verzi fronty ze zadání 2.2. Vytvořte generické rozhraní `IQueue` s metodami `enqueue(T item)`, `dequeue()`, `isFull()`, `isEmpty()` a `clear()`. Ve třídě `Queue`, která bude generická a bude implementovat rozhraní `IQueue`, tyto metody naprogramujte. Pro uchovávání prvků použijte generický `ArrayList`. Fronta bude mít omezenou velikost, kterou v konstruktoru uložte do atributu `size`. Naprogramujte také metodu `reorder()`, která přeskládá frontu od nulté pozice, aby `ArrayList` nenabýval větší velikosti než zadaný atribut `size`.

Testování

Vytvořte si instanci fronty s typovým parametrem `String` a velikostí 3. Zavolejte metody `isFull()` a `isEmpty()`, vložte tři prvky a znovu zavolejte metody `isFull()` a `isEmpty()`. Prvky postupně z fronty odeberte a ověřte, že se odebírají ve správném pořadí. Zkuste také, že nedojde

k přetečení a podtečení fronty. Nakonec si vytvořte druhou instanci fronty s typovým parametrem `Double` a frontu otestujte podobným způsobem.

Omezení: fronta má omezenou velikost zadanou v konstruktoru.

Obtížnost: 2 - mírně pokročilá.

Zahrnuté oblasti: fronta, generické typy, ArrayList, rozhraní.

5.3 Zadání 3.3 - spojový seznam a iterátor

Specifikace

Naprogramujte generický jednosměrný spojový seznam, který bude obsahovat vlastní iterátor. Vytvořte třídu `MyLinkedList<T>`, která bude implementovat rozhraní `Iterable<T>`. Třída bude obsahovat privátní atributy `size` a `count`, které uchovávají údaje o maximální a aktuální velikosti seznamu a atributy `first` a `last`, které budou datového typu `Node`. Třídu `Node` zde nadefinujte jako vnořenou třídu, jejími atributy budou `item` s generickým datovým typem `T` a `next` datového typu `Node`. Třída `MyLinkedList` bude obsahovat metodu pro přidání prvku na konec seznamu a metody pro odstranění a vrácení prvku na základě indexu.

Nakonec zde musíte naimplementovat metodu `iterator()` danou rozhraním `Iterable`. V té vraťte nový objekt datového typu `Iterator<T>`, který zde nadefinujete jako anonymní třídu. Iterator obsahuje metody `hasNext()`, `next()` a `remove()`. Metoda `hasNext()` vrací *true*, pokud existuje další prvek. Metoda `next()` vrátí další prvek a metoda `remove()` je nepovinná a není potřeba ji implementovat.

Testování

Vytvořte si instanci spojového seznamu o určité velikosti a s daným typovým parametrem, např. `Integer`. Do seznamu vložte několik prvků a vypište si je pomocí cyklu foreach. Ze seznamu si vraťte některý prvek a také vyzkoušejte odebírání ze seznamu. Nakonec vyzkoušejte také průchod seznamu za pomoci Iteratoru a cyklu `while`.

Omezení: spojový seznam má omezenou velikost zadanou v konstruktoru.

Obtížnost: 3 - pokročilá

Zahrnuté oblasti: spojový seznam, generické typy, Iterable, Iterator, vnořené třídy.

5.4 Zadání 3.4 - HashMap

Specifikace

Naprogramujte vlakový jízdní řád za pomoci generické Mapy. Vytvořte třídu `Train` s atributy `number`, `type`, `name`, `timeOfDeparture` a seznamem `stops`. Vytvořte odpovídající konstruktory tak, abyste zadali všechny atributy, pouze jméno bude nepovinné. Vytvořte také get a set metody, metodu pro přidání zastávky, pro výpis všech zastávek a nakonec metodu `toString()` pro výpis.

Další třídou bude `TrainSchedule`, která bude obsahovat generickou Mapu, do které vytvořte instanci `HashMap` s typovými parametry `Integer` a `Train`. Jako klíč se bude používat číslo vlaku. Vytvořte metody pro přidání vlaku, získání a odstranění vlaku na základě čísla a metody pro výpis všech vlaků a všech klíčů Mapy, tedy všech čísel vlaků.

Testování

Vytvořte si instanci třídy `TrainSchedule` a několik vlaků. K jednotlivým vlakům přidejte různé zastávky a vlaky přidejte do jízdního řádu. Nechte si vypsat jednotlivá čísla vlaků, poté některý vlak odeberte a jiný vlak si vraťte a vypište všechny jeho zastávky. Nakonec si nechte všechny vlaky z jízdního řádu vypsat. Jednotlivé výpisy zkontrolujte a ověřte, že vám vše funguje správně.

Omezení: při vytváření vlaku je nutno zadat všechny odpovídající atributy; k vlaku lze přistupovat pouze na základě jeho čísla.

Obtížnost: 2 - mírně pokročilá

Zahrnuté oblasti: HashMap, ArrayList, procházení kolekce.

5.5 Zadání 3.5 - EnumMap

Specifikace

Naprogramujte školní rozvrh, který budete ukládat do Mapy, kde klíči budou jednotlivé prvky výčtového typu. Vytvořte si tedy výčtový typ `Days` se všemi dny. Dále si vytvořte třídu `Subject` s atributy `name` a `time`, odpovídajícím konstruktorem, get a set metodami a metodou `toString()`. Nakonec ještě naprogramujte třídu `SchoolSchedule` s atributem typu `EnumMap`, který bude používat jako klíč prvky `Days` a jako hodnotu bude mít `List` předmětů. Vytvořte metodu pro přidání jednodenního rozvrhu, metodu, která vypíše jednotlivé dny v rozvrhu a také metodu, která vypíše celý rozvrh. Nakonec ještě vytvořte metodu pro výpis rozvrhu daného dne určeného parametrem a metodu, která spočítá počet předmětů v celém rozvrhu.

Testování

Vytvořte si jednotlivé ArrayListy s předměty odpovídajícími jednodennímu rozvrhu. Poté si vytvořte instanci třídy `SchoolSchedule` a dané ArrayListy přidejte do rozvrhu k určitému dni. Nyní zavolejte metody s jednotlivými výpisy a vše zkontrolujte. Jako poslední si nechte spočítat počet předmětů a opět výpis zkontrolujte.

Omezení: při vytváření předmětu je nutno zadat jeho jméno a čas; neukládá se délka daného předmětu a neexistuje kontrola, zda se předměty nepřekrývají.

Obtížnost: 2 - mírně pokročilá.

Zahrnuté oblasti: výčtový typ, EnumMap, ArrayList, procházení kolekce.

5.6 Zadání 3.6 - Queue

Specifikace

Naprogramujte simulaci pošty s využitím předdefinované fronty. Vytvořte třídu `Customer` s atributy `name` a `timeNeeded`, který určuje, kolik času potřebuje daný zákazník na obsloužení. Zajistěte všechny potřebné věci pro nastavování hodnot a pro výpis zákazníka. Další třídou bude `Counter` s celočíselným atributem `id`, atributem `customer` datového typu `Customer` a pomocnými atributy `currentTime` a `empty`. V konstruktoru nastavte `id` podle parametru, `currentTime` na 0 a `empty` na *true*. Vytvořte get metodu pro atribut `empty` a set metodu pro atribut `customer`, ve kterém také vynulujete aktuální čas a atribut `empty` nastavíte na *false*. Hlavní metodou bude `servingCustomer()`, ve které inkrementujete atribut `currentTime` a ověříte zda je zákazník již obsluhován potřebný čas. Jestliže ano, tak to vypíšete na konzoli a atribut `empty` nastavíte na *true*. Nakonec ještě vytvořte metodu `remainingTime()` vracející zbývající potřebný čas pro zákazníka.

Hlavní třídou bude `PostOffice`, která bude obsahovat `List` přepážek a frontu zákazníků. Pro frontu použijte generickou třídu `Queue`. V konstruktoru vytvoříte zadaný počet přepážek. Samozřejmostí je metoda pro přidání zákazníka do fronty. Další metodou bude `servingCustomers(int time)`, která bude v cyklu od 0 do parametru `time` volat na všechny přepážky metodu `servingCustomer()` a v případě, že je přepážka prázdná, tak přidá dalšího zákazníka z fronty. Naprogramujte také metody, které vrátí počet zákazníků ve frontě, vrátí zákazníka, který je další na řadě a metodu, která vypíše všechny zákazníky z fronty. Poslední metoda, kterou naprogramujete, bude vracet přibližný čas k obsloužení všech zákazníků. Ten vypočítejte jako součet potřebného času všech zákazníků vydělený počtem překážek. Vypočítanou hodnotu vraťte jako celočíselnou hodnotu zaokrouhlenou nahoru.

Testování

Vytvořte novou poštu se třemi přepážkami a devět zákazníků s časy: 20, 10, 30, 15, 25, 12, 28, 16 a 24. Nyní obsluhujte zákazníky po dobu 30-ti jednotek. Měli by vám zůstat 3 zákazníci ve frontě s odhadovanou dobou 30. Zákazníky si také vypište. Obsluhujte zákazníky dalších 15 časových jednotek a opět si vypište stav pošty pomocí daných metod. Měl by vám zůstat jeden zákazník ve frontě a odhadovaný čas k dokončení 15 jednotek. Proveďte poslední obsloužení po dobu 25-ti jednotek a ověřte, že je fronta prázdná a potřebný čas 0.

Omezení: při vytváření zákazníka je nutno zadat jméno a potřebný čas; vypočítaný zbývající čas je pouze odhadem.

Obtížnost: 3 - pokročilá

Zahrnuté oblasti: Queue, ArrayList, procházení kolekce.

5.7 Zadání 3.7 - ohraničení typových parametrů

Specifikace

Vytvořte abstraktní třídu `Person` s chráněnými atributy `name` a `age` a abstraktní metodou `print()`. Nyní vytvořte třídu `Employee`, která bude dědit z `Person` a navíc bude implementovat generické rozhraní `Comparable`. Přidejte privátní atribut `salary`. Všechny atributy nastavte v konstruktoru a doimplementujte potřebné metody. Porovnání proveďte podle atributu `name`. Další třídou dědící z `Person` bude třída `Student`, která bude opět implementovat rozhraní `Comparable`. Přidejte atribut `login` a podobně jako ve třídě `Employee` naprogramujte konstruktor a potřebné metody. Tentokrát se bude porovnávat podle atributu `login`. Poslední třídou dědící z `Person` bude `Unemployed`, ve které pouze nastavte v konstruktoru atributy a doimplementujte metodu `print()`.

Nakonec si vytvořte třídu `PersonAgenda`, která bude generická a bude přijímat jako typový parametr pouze třídy, které dědí z `Person` a zároveň implementují rozhraní `Comparable`. Ve třídě bude pouze seznam osob a metody pro přidání, vypsání a seřazení.

Testování

Vytvořte si dvě instance třídy `PersonalAgenda`, jednu s typovým parametrem `Student` a druhou s parametrem `Employee`. Vytvořte také několik studentů a zaměstnanců a přidejte je do odpovídajících agend. Můžete si také vyzkoušet, že vám nepůjde vložit zaměstnance do agendy se studenty a naopak. Vypište si neseřazené studenty, poté je seřaďte a znovu vypište. To samé proveďte i se zaměstnanci a ověřte, že se každá agenda seřazuje podle správného atributu. Nakonec zkuste ještě vytvořit instanci `PersonalAgenda` s typovým parametrem `Unemployed`. Zjistíte, že to nelze, protože třída `Unemployed` neimplementuje rozhraní `Comparable`, a proto by nebylo možné agendu seřadit.

Omezení: do agendy lze ukládat jen třídy dědící z `Person` a implementující `Comparable`; při vytváření jednotlivých konkrétních osob je nutno zadat odpovídající parametry.

Obtížnost: 2 - mírně pokročilá.

Zahrnuté oblasti: generické typy a jejich ohraničení, procházení kolekce, ArrayList, Comparable, dědičnost.

5.8 Zadání 3.8 - žolíci

Specifikace

Vyzkoušejte si práci s žolíky. Vytvořte si opět hierarchii tříd na téma tvary, čili třídu `Shape`, která tentokrát nebude abstraktní a bude mít chráněný atribut `name`, který nastavíte v konstruktoru. Třída bude mít jednu metodu `draw()`, která pouze vypíše, že se vykresluje tvar s daným názvem. Z této třídy budou dědit `Circle` a `Rectangle` s obdobnou implementací a z třídy `Rectangle` bude dále dědit třída `Square`.

Poslední třídou bude `ShapesOperations`, která bude obsahovat generický seznam `shapesWithGenerics` a negenerický seznam `shapesWithoutGenerics`. Vytvořte jednu metodu pro přidávání, která přidá přijatý tvar do obou seznamů a poté ke každému seznamu metodu get. Následující metody budou statické, bude to metoda `drawAllShapes(List<Shape> shapes)`. Metoda pouze projde přijatý seznam a zavolá na každý objekt metodu `draw()`. Obdobně bude existovat metoda `drawAllCircles(List<Circle> circles)`. U těchto metod můžeme přijmout pouze určitý `List`, my ale chceme mít jednu metodu, která bude zvládat všechny Listy se stejným předkem. Napište tedy další metodu `drawAllShapesWithWildcard(List<? extends Shape> shapes)`. V této metodě při průchodu cyklem foreach si vyzkoušejte použít jako datový typ `?`, zjistíte, že to nelze. V poslední metodě jsme použili ohraničení za pomoci `extends`, můžeme k ohraničení použít i klíčové slovo `super`. Vytvořte tedy metodu `drawAllSquareAncestors(List<? super Square> shapes)`. Poslední statickou metodou bude `addShapeToList(Shape shape, List<? extends Shape> shapes)`. Pokuste se v této metodě objekt do daného seznamu přidat. Tato operace je zakázaná, protože nemůžeme zaručit, že nebudeme např. volat metodu, kde bychom chtěli přidat `Rectangle` do seznamu `Circle`.

Testování

Vytvořte si generický seznam `shapes`, do kterého uložíte instance `Shape`, `Circle` a např. `Rectangle`. Dále si vytvořte druhý generický seznam `circles`, do kterého uložte několik kruhů. Zavolejte statickou metodu `drawAllShapes()`, které předáte seznam `shapes`. Poté zkuste to samé, ale předat seznam `circles` - to už by nemělo projít. Nyní vyzkoušejte metodu `drawAllCircles()` a postupně ji předejte oba dva seznamy. Výsledek bude obdobný, proto zavolejte metodu `drawAllShapesWithWildcards()`, které již můžete oba dva seznamy předat bez problému. Poslední statickou metodou, kterou si vyzkoušejte je `drawAllSquareAncestor()`. K tomu si vytvořte podobně jako na začátku seznamy `rectangles` a `squares`. Této metodě postupně předejte všechny vámi vytvořené seznamy a zjistěte, ve kterých případech je vše v pořádku a ve kterých nikoliv.

Nyní si ještě vyzkoušejte nestatickou část třídy. Vytvořte si proto instanci `ShapesOperations` a přidejte do ní nějaké tvary. Vraťte si z této třídy generický seznam a uložte ho do negenerického lokálního Listu. Vyzkoušejte si průchod tímto Listem za pomoci cyklu foreach. Nyní si vraťte negenerický seznam a uložte jej do proměnné typu `List<Shape>`. Opět projděte celý seznam. V obou případech dostanete od kompilátoru warning. Zkuste nyní znovu vrátit oba dva seznamy a pokaždé je uložit do proměnné typu `List<?>` a zase si vyzkoušet foreach. Všimněte si, že tentokrát je už vše bez warningů.

Omezení: při vytváření tvarů je nutno zadat jejich jména; při vkládání tvaru do třídy `ShapesOperation` se objekt uloží do obou seznamů.

Obtížnost: 3 - pokročilá.

Zahrnuté oblasti: zástupný znak ?, ohraničené generické typy, ArrayList, průchod kolekcemi, dědičnost.

6 Zpracování výjimek

6.1 Zadání 4.1 - běhové výjimky

Specifikace

Vyzkoušejte si v jakých situacích se vyvolávají některé běhové výjimky. Vytvořte si třídu `Person` s atributy `listOfFriends` typu `List<String>`, pole `arrayOfHobbies` typu `Object[]`, `numberOfHobbies` typu `int` a `age` typu `Object`. V konstruktoru vytvořte do `arrayOfHobbies` pole typu `String` o velikosti 3. Vytvořte setter pro atribut `age` a metodu `printAge()`, která přetypuje atribut `age` na `String` a vypíše ho na konzoli. Další metodou bude `createFriends()`, která vytvoří do atributu `listOfFriends` nový seznam a vloží několik přátel. Metoda `listAllFriends()` všechny tyto přátele vypíše. V další metodě `listAllFriendsWithIterator()` vypište přátele za pomoci Iteratoru, ale cyklus nastavte tak, že se bude vypisovat desetkrát. Poslední metodou bude `addHobby(Object hobby)`, která do pole `arrayOfHobbies` vloží objekt na pozici `numberOfHobbies`. Tento atribut poté inkrementujte.

Nakonec si vytvořte rozhraní `IPrinter` s metodou `print()`. Naše třída `Person` bude toto rozhraní implementovat. Některá vývojová prostředí vám automaticky metody z rozhraní doplní a v jejich těle se bude pouze vyvolávat výjimka `UnsupportedOperationException`. Pokud tohle za vás vaše prostředí neudělalo, tak to dopište manuálně.

Testování

Vytvořte si instanci třídy `Person` a jako první si zkuste vyvolat výjimku `ArrayStoreException`. Zavolejte metodu `addHobby()` s parametry například sports a friends. Metoda přijímá jako parametr datový typ `Object`, a proto zkuste zavolat metodu s parametrem např. `new Integer(15)`. Další výjimka, se kterou se můžete setkat poměrně často, je `IndexOutOfBoundsException`. Vyvoláte ji tak, že se pokusíte vložit prvek na pozici mimo rozsah alokovaného pole. Zavolejte tedy metodu `addHobby()` celkem čtyřikrát. Další výjimkou, kterou si vyzkoušejte, je `ClassCastException`. Zavolejte metodu `setAge()` s parametrem `new Integer(20)` a poté metodu `printAge()`. Program se snaží přetypovat číslo na `String` a vyvolá výjimku.

Asi nejčastější výjimkou je `NullPointerException`, tu vyvoláte zavoláním metody `listAllFriends()`, pokud jste předtím nevolali metodu `createFriends()`. Další výjimkou je `NoSuchElementException`. Zavolejte metodu `createFriends()` a poté `listAllFriendsWithIterator()`. Výjimka je vyvolána, pokud se snažíme zavolat další prvek z iteratoru, ale žádný další prvek neexistuje. Nakonec ještě zavolejte metodu `print()` a ověřte, že se opravdu vyvolá výjimka `UnsupportedOperationException`.

Omezení: žádné.

Obtížnost: 2 - mírně pokročilá.

Zahrnuté oblasti: běhové výjimky.

6.2 Zadání 4.2 - běhové výjimky a jejich odchytávání

Specifikace

Vyzkoušejte si vyvolávání běhových výjimek a jejich odchytávání. Vytvořte třídu `Person` s atributy `firstName`, `lastName`, `age`, polem `friends` typu `String` a dvěma celočíselnými hodnotami `position` a `size`. Vytvořte get a set metody pro atributy `firstName`, `lastName` a `age`. V set metodách pro hodnoty typu `String` zajistěte, že předaná hodnota nebude *null* nebo prázdný řetězec a u atributu `age` zajistěte, že parametr nebude záporný. V takových případech vyvolejte výjimku `IllegalArgumentException` s odpovídající zprávou. V konstruktoru budete také nastavovat atributy `firstName`, `lastName` a `age` a ošetřovat je stejným způsobem jako v set metodách. Další metodou bude `createArrayOfFriends(int size)`, která vytvoří pole o zadané velikosti. Pokud je velikost záporná, tak se vyvolá výjimka `NegativeArraySizeException`. Poslední metodou této třídy bude `addFriend(String friend)`, která přidá přítele do pole. Pokud je pole plné, tak se vyvolá výjimka `IndexOutOfBoundsException`.

Druhou třídou bude `Calculator`, kde nadefinujete metody pro sčítání, odečítání, násobení, dělení, n-tou mocninu a druhou odmocninu. U dělení ošetřete, že nelze dělit nulou a případně vyvolejte výjimku `ArithmeticException`. U odmocniny ošetřete stejnou výjimkou, že číslo pod odmocninou nesmí být záporné.

Testování

V metodě `main()` si vytvořte v bloku `try` instanci třídy `Person`, která bude v pořádku. Poté si zkuste vytvořit další, které předáte postupně hodnotu *null*, prázdný řetězec a záporný věk. Vyzkoušejte si také toto nastavování přes set metody. Dále v tomto bloku `try` zavolejte metodu `createArrayOfFriends()` napřed se záporným parametrem a potom s parametrem 2. Následně si vyzkoušejte přidat do pole tři záznamy. Za tímto blokem `try` budete mít 3 bloky `catch`, ve kterých budete odchytávat jednotlivé výjimky. V jednom bloku `catch` si nechte vypsat zprávu pomocí metody `getMessage()`, v dalším si nechte vypsat samotnou výjimku pomocí `toString()` a v posledním bloku zavolejte metodu `printStackTrace()` na odchytnutou výjimku.

Nakonec si vytvořte druhý blok `try`, ve kterém vytvoříte instanci třídy `Calculator` a vyzkoušíte jednotlivé metody s korektními i nekorektními parametry. V bloku `catch` odchytávejte výjimku `ArithmeticException`.

Omezení: kalkulačka pracuje s datovým typem `double` a základní operace přijímají dva parametry.

Obtížnost: 2 - mírně pokročilá.

Zahrnuté oblasti: běhové výjimky, blok try-catch.

6.3 Zadání 4.3 - vlastní a běhové výjimky

Specifikace

Vytvořte program počítající pojištění osob a jejich vozidel. Vytvořte si jednoduchou třídu `Person` s atributem `age` a get metodou. Hodnotu nastavte v konstruktoru, v případě zadání záporného čísla vyvolejte výjimku `IllegalArgumentException`. Nyní si vytvořte vlastní výjimku `AgeException`, která bude dědit z `Exception` a bude obsahovat bezparametrický a parametrický konstruktor. V parametru budete přijímat zprávu a v těle zavoláte konstruktor předka.

Dále budete potřebovat rozhraní `IVehicle` s metodou `countInsurance(Person p)`, která vrací hodnotu typu `int` a vyvolává výjimku `AgeException`. Toto rozhraní budou implementovat třídy `Car` a `Motorcycle`. Třída `Car` bude mít atribut `horsePower`, který nastavíte v konstruktoru a také vytvoříte get metodu. V metodě `countInsurance()` napřed ověřte, zda není osoba mladší než 18 let, případně vyvolejte odpovídající výjimku, a dále vypočtěte pojištění. Pokud je `horsePower` menší než 200, tak bude pojištění 3000, jinak bude pojištění 5000. Také pokud je osoba mladší 21 let, tak zvyšte cenu o 30%. Třída `Motorcycle` bude mít atribut `cubicCapacity`, který opět budete nastavovat v konstruktoru a vytvoříte k němu get metodu. Metoda `countInsurance()` bude také obdobná. Osoba tentokrát nesmí být mladší 21 let a pokud je `cubicCapacity` menší než 500, tak bude pojištění stát 1500, jinak bude stát 2500.

Testování

Vytvořte blok try-catch, odchytávat budete výjimky `IllegalArgumentException` a `AgeException`. V bloku `try` si vytvořte čtyři osoby s věkem -21, 19, 17 a 30, dvě auta s výkonem 200 a 400 a také dvě motorky s objemem 250 a 750. Vyzkoušejte si, že se vám odchytne výjimka pro záporný věk a poté daný řádek zakomentujte. Vyzkoušejte si různé kombinace osob a vozidel a zkontrolujte, že se odchytí ve správném případě `AgeException` a také, že se pojištění počítá správně.

Omezení: atributy všech objektů je nutno nastavovat při jejich vytváření.

Obtížnost: 2 - mírně pokročilá.

Zahrnuté oblasti: běhové výjimky, vlastní výjimky, blok try-catch, klauzule throws, rozhraní.

6.4 Zadání 4.4 - vlastní výjimky a způsoby odchytávání

Specifikace

Vyzkoušejte si různé způsoby odchytávání výjimek na příkladu kavárny. Jako první si vytvořte vlastní výjimky. První bude `TemperatureException` s atributem `temperature`, který nastavíte v konstruktoru a umožníte k němu přístup pomocí get metody. Z této výjimky budou dědit `TooColdException` a `TooHotException`. Poslední výjimkou bude `TooLongException`.

Vytvořte třídu `CupOfCoffee` s atributem `temperature`, odpovídající get a set metodou. Tento atribut nastavujte v konstruktoru. Další třídou bude `Person` s atributy `lowTemperature`, `highTemperature` a `waitingTime` určující hraniční hodnoty pro vyvolání výjimek. Tyto

atributy nastavujte v konstruktoru a vytvořte k nim set metody. Posledním atributem bude `coffee` datového typu `CupOfCoffee`. Tento atribut budete nastavovat v metodě `serveCoffee(CupOfCoffee c, int serveTime)`, která bude vyhazovat výjimku `TooLongException`. V metodě přiřaďte kávu do privátního atributu a porovnejte `serveTime` s `waitingTime`, v případě, že je `serveTime` větší, vyvolejte výjimku. Poslední metodou bude `drinkCoffee()`, která bude vyhazovat výjimky `TooColdException` a `TooHotException`. V těle metody pouze ověřte teplotu kávy a případně vyvolejte odpovídající výjimky.

Poslední třídou bude `CoffeeHouse`, která bude obsahovat pouze několik statických metod `serveCustomer1(Person p, CupOfCoffe c)` až `serveCustomer7(Person p, CupOfCoffe c)` s různou implementací. Ve všech metodách použijte pro parametr `serveTime` statickou metodu `Random()` ze třídy `Math`, která bude generovat čísla 1-10. V první metodě v jednom bloku `try` zavolejte na danou osobu `serveCoffe()` s příslušnými parametry a poté `drinkCoffe()` a vypište, že vše proběhlo v pořádku. Odchyťte postupně výjimky `TooLongException`, `TooColdException` a `TooHotException`. V druhé metodě si udělejte dva bloky `try`. V jednom se bude řešit obdobným způsobem servírování kávy a v druhém pití kávy. Třetí metoda bude stejná jako první, ale výjimky `TooColdException` a `TooHot Exception` odchyťte v jednom bloku catch.za pomoci operátoru |. Čtvrtá metoda bude mít opět stejné tělo, ale tentokrát bude odchytávat výjimky `TooLongException` a `TemperatureException`. V páté metodě odchytávejte výjimky v pořadí `TooLongException`, `TooHotException` a `TemperatureException`. V šesté metodě budete odchytávat v pořadí `TooLongException`, `TooColdException`, `TooHotException` a navíc přidáte blok `finally`, ve kterém vypíšete teplotu kávy. A v poslední metodě budete odchytávat pouze `TemperatureException` a výjimku `TooLongException` nadeklarujete v hlavičce metody pomocí klauzule `throws`. V každém bloku `catch` vypište na obrazovku odpovídající výpisy. V případě, že není jasné jestli se jedná o `TooColdException` nebo `TooHotException`, výpis rozlište na základě atributu `temperature` dané výjimky.

Testování

Vytvořte si osobu a šálek kávy s vámi zvolenými parametry, ty vhodně měňte, abyste vyvolali všechny tři teplotní stavy, zavolejte tedy třikrát metodu `serveCustomer1()`. Sledujte a kontrolujte, které výjimky se vám vyvolaly. Také se mohlo stát, že se vám vyvolala výjimka `TooLongException`, která závisí na metodě `Random()`. Pokud se vám ani v jednom případě nevyvolala, tak nastavte čekací dobu zákazníka na 0 a znovu zavolejte metodu `serveCustomer1()`. Všimněte si, že po vyvolání této výjimky už nedojde na metodu `drinkCoffee()`. Nyní zavolejte metodu `serveCustomer2()`, ve které se již provede i metoda `drinkCoffee()`. Poté nastavte atribut `serveTime` na číslo větší než 10, aby se vám výjimka `TooLongException` už nevyvolávala.

Otestujte metodu `serveCustomer3()` tak, aby byla teplota jednou příliš nízká a podruhé příliš vysoká. To stejné proveďte i u metody `serveCustomer4()`. U metody `serveCustomer5()` nastavte, aby teplota byla příliš velká a všimněte si, který blok `catch` se zavolá. Metodu `serveCustomer6()` volejte tak, abyste jednou vyvolali výjimku a jednou vše

proběhlo v pořádku. Všimněte si, že v obou případech se provede i blok `finally`. Poslední metodu `serveCustomer7()` budete muset obalit do bloku try-catch. Vyzkoušejte její volání tak, abyste vyvolali `TooLongException`, poté teplotní výjimku a nakonec, aby metoda prošla bez chyby.

Omezení: výjimka `TooLongException` je vyvolávána na základě hodnoty z metody `Random()`, která může být při každém spuštění programu jiná.

Obtížnost: 3 - pokročilá.

Zahrnuté oblasti: vlastní výjimky, blok try-catch, blok try-catch-finally, klauzule throws.

6.5 Zadání 4.5 - vlastní výjimky

Specifikace

Naprogramujte bankovní účet s vyvoláváním vlastních výjimek. Vytvořte si výjimky `NotEnoughMoneyException`, `WrongPinException`, `ThreeTimesWrongPinException` a `BlockedAccountException`, ve které předkovi předáte zprávu, že je účet zablokovaný.

Vytvořte si třídu `Account`, která bude mít privátní atributy `numberOfAccount`, `balance`, `pin` a pomocné atributy `numberOfWrongPinsEntered` a `blocked` typu `boolean`. Číslo účtu a pin nastavte v konstruktoru. Zůstatek na účtu bude po založení 0. Pro tento atribut vytvořte také get metodu. Vytvořte metody pro vložení peněz na účet a pro výběr z účtu. Obě dvě metody budou vyhazovat výjimku `BlockedAccountException` v případě, že je účet zablokovaný. Pro výběr peněz je také nutno zadat pin. Ověřte, jestli je na účtu dostatek peněz a jestli byl správně zadán pin a případně vyvolejte výjimky. Na správné zadání pinu má uživatel tři pokusy, poté se vyvolá výjimka a účet se zablokuje. V této třídě odchytávejte na správných místech pouze výjimky týkající se pinu, všechny ostatní deklarujte v hlavičce metody.

Testování

Vytvořte si nový účet. V bloku `try` na něj vložte 1000 a následně 500 vyberte. Odchyťte potřebné výjimky a v bloku `finally`, stejně jako ve všech následujících případech, vypište zůstatek účtu. V dalším bloku `try` se pokuste vybrat 1500. Napište další blok `try`, ve kterém zadejte třikrát špatný pin. V posledním bloku `try` se ještě pokuste vybrat 200 a ověřte si, že je účet opravdu zablokovaný. Zkontrolujte, že se vám všechny výjimky správně vyvolávají.

Omezení: číslo účtu a pin je nutno zadat v konstruktoru; účet již nelze odblokovat.

Obtížnost: 3 - pokročilá.

Zahrnuté oblasti: vlastní výjimky, blok try-catch, blok try-catch-finally, klauzule throws.

6.6 Zadání 4.6 - vlastní výjimky 2

Specifikace

Naprogramujte simulaci cesty autem s vyvoláváním výjimek. Vytvořte výjimky `EndTripException`, `NegativeSpeedException` a `SpeedLimitException` předávající

vždy konstruktoru předka odpovídající zprávu a výjimku `EngineException`, které v konstruktoru předáte parametr `message` a ten předáte konstruktoru předka.

První třídou bude `Road` s atributy `distance` a `speedLimit`, které nastavíte v konstruktoru a také pro ně vytvoříte get a set metody. Další třídou bude `Trip` s privátní kolekcí cest `roads`, atributem `currentRoad` typu `int` uchovávající informaci, na které cestě z kolekce jsme, atributem `distancePassedFromRoad` a atributem `finished` typu `boolean`. Pro všechny atributy vytvořte get metody a pro `currentRoad` a `distancePassedFromRoad` ještě set metody. Vytvořte také metodu pro přidání cesty do kolekce. Třída bude mít dále metody `distancePassed(double distance)` a `moveToNextRoad()`. Metoda `moveToNextRoad()` zkontroluje jestli jsme na konci výletu a případně nastaví další cestu z kolekce jako aktuální. Metoda `distancePassed()` upraví atribut `distancePassedFromRoad()` a pokud jsme na konci cesty, tak nastaví další cestu.

Poslední třídou bude `Car` s atributy `engineStarted` typu `boolean`, `currentSpeed` typu `int` a `trip` typu `Trip`. Vytvořte set metodu pro atribut `trip` a get metodu pro atribut `currentSpeed`. Naprogramujte metody `startEngine()` a `stopEngine()` vyhazující výjimky `EngineException`. Výjimku vyvolejte v případech, že se snažíte nastartovat již nastartovaný motor nebo vypnout již vypnutý motor a nebo vypnout motor pokud je rychlost auta větší než 0. Další metodou bude `increaseSpeed(int amount)` vyhazující `EngineException` a `SpeedLimitException`. Výjimky vyvolejte v případě, že není motor nastartovaný nebo nová rychlost překročila povolený limit dané cesty. Obdobnou metodou bude `decreaseSpeed(int amount)` vyhazující výjimky `EngineException` a `NegativeSpeedException` v případech, že motor neběží a že by rychlost klesla pod 0. Další metodou bude `drive(int minutes)` vyhazující `SpeedLimitException`, `EndTripException` a `EngineException`. Metoda napřed ověří, jestli jsme nedorazili do cíle výletu a jestli máme nastartovaný motor. Poté metoda bude v cyklu od 0 do zadaného parametru `minutes` ověřovat, jestli jsme dojeli na konec výletu, v případě, že ano vypíše informaci na konzoli a poté zastaví auto a vypne motor. Pokud ještě nejsme v cíli, tak zavolá metodu `distancePassed()` na atribut `trip` a jako parametr předá uraženou vzdálenost za jednu minutu (vypočítá se jako rychlost/60). Ještě před zavoláním metody `distancePassed()` je nutno ověřit, jestli jsme nepřekročili rychlostní limit. Poslední metodou bude `printCurrentState()`, která pouze vypíše informace o současné rychlosti, uražené vzdálenosti, počtu cest do konce a momentálním rychlostním limitu.

Testování

Vytvořte si nový `Trip` a vložte do něho postupně tři cesty. První bude mít délku 50 a rychlostní limit 50, druhá délku 30 a limit 90 a poslední délku 100 a limit 130. Nyní si vytvořte nové auto a předejte mu vytvořený `Trip`. Napřed zkuste vypnout motor a přesvědčte se, že se vyvolá výjimka. Následně to samé vyzkoušejte se zvýšením rychlosti a metodou `drive()`. Nyní nastartujte motor, nastavte rychlost na 30 a jeďte po dobu třiceti minut. Správně odchytávejte výjimky a vždy v bloku `finally` si vypište aktuální stav. V tuto chvíli by vám měly zbývat do konce 3 cesty a 35 kilometrů do konce aktuální cesty. Zvyšte rychlost o 30, to by mělo vyhodit výjimku, a proto následně snižte rychlost o 10 a jeďte po dobu 10-ti minut. Nyní by vám měly zbývat 3 cesty a 26,6 km. Jeďte po

dobu dalších 50-ti minut. Do cíle zbývají 2 cesty a 14,99 km. Zvyšte rychlost o 30 a jeďte 20 minut. Zbývá poslední cesta a 89,33 km. Zrychlete o dalších 30 a jeďte 20 minut. Do cíle zbývá 52,66 km. Jeďte posledních 40 minut, během kterých byste měli dojet do cíle. Nakonec zkuste ještě jednou zavolat metodu `drive()`, ta by vám neměla dovolit další jízdu. Všechny vaše výpisy si zkontrolujte a porovnejte s tímto popisem.

Omezení: pokud do konce cesty zbývá menší vzdálenost než je předaná parametrem metodě `distancePassed`, tak se přejde pouze na další cestu a nadbytečné kilometry se nepřenáší.

Obtížnost: 4 - velmi pokročilá.

Zahrnuté oblasti: vlastní výjimky, blok try-catch, blok try-catch-finally, klauzule throws.

6.7 Zadání 4.7 - rozhraní Closeable a blok try()

Specifikace

Vyzkoušejte si práci s novým blokem `try()`, který uvolňuje zdroje a také zjistěte, jestli se zdroje uvolňují až po zpracování výjimky nebo před. Pro vyzkoušení si naprogramujte jednoduchou třídu `ConsolePrinter`, která musí implementovat rozhraní `Closeable`. V konstruktoru vypište do konzole, že se vytváří nová tiskárna. Naimplementujte metodu `print(String line)`, která vypíše parametr na konzoli. Pokud je parametr *null*, tak vyvolá výjimku `IllegalArgumentException` s odpovídající zprávou. V metodě `close()`, která je definována rozhraním pouze vypište na konzoli, že se tiskárna ukončuje.

Testování

Vytvořte si instanci tiskárny a v klasickém bloku `try` zavolejte metodu `print()` s parametrem např. "Hello". Blok doplňte o část `finally`, ve které budete volat metodu `close()`. Dále si vytvořte druhou instanci a vytvořte tentokrát blok try-catch-finally, kde budete odchytávat výjimku `IllegalArgumentException`. Metodu zavolejte s parametrem *null*. V bloku `finally` opět volejte metodu `close()` a zjistěte, kdy se tiskárna uzavírá.

Nyní si vyzkoušejte nový blok `try`, kde v závorkách vytvoříte novou instanci tiskárny a v následném bloku kódu zavoláte metodu s korektním parametrem. Všimněte si, že se automaticky zavolá metoda `close()` a také, že mimo blok `try` již nemáte přístup k proměnné tiskárny. Nakonec si vytvořte nový blok `try()` stejným způsobem jako poprvé, tentokrát jej ale doplňte o blok `catch` s odchytáváním výjimky `IllegalArgumentException`. Metodu nyní zavolejte s parametrem *null* a zjistěte, kdy se tiskárna ukončí a porovnejte to s klasickým blokem try-catch-finally.

Omezení: žádné.

Obtížnost: 2 - mírně pokročilá.

Zahrnuté oblasti: běhové výjimky, blok try-catch-finally, blok try() s uvolňováním zdrojů.

7 Datové proudy

7.1 Zadání 5.1 - čtení z klávesnice

Specifikace

Naprogramujte jednoduchou kalkulačku, která bude načítat argumenty z klávesnice. Vytvořte si třídu `Calculator` s jedním privátním atributem typu `BufferedReader` a veřejnou metodou `execute()`, ve které do atributu uložíte instanci `BufferedReader(new InputStreamReader(System.in))`. Dále v této metodě vytvořte menu s možnostmi: sčítání, odčítání, násobení, dělení a konec programu. Ostatní metody budou privátní a budou obstarávat jednotlivé matematické operace od načtení argumentů po zobrazení výsledku. Načítané hodnoty jsou typu `String`, a proto je správným způsobem přetypujte. Odchyťte také potřebné výjimky. Při ukončení programu nezapomeňte uzavřít `BufferedReader`.

Testování

Vytvořte si instanci `Calculator` a zavolejte metodu `execute()`. Vyzkoušejte si jednotlivé části programu, zda vám správně počítají hodnoty a také, že se vám správně odchytávají výjimky, čili že vám program nespadne, ale pouze vypíše chybovou hlášku a dále pokračuje.

Omezení: početní metody pracují s dvěma argumenty typu `double`.

Obtížnost: 2 - mírně pokročilá.

Zahrnuté oblasti: načítání z klávesnice, BufferedReader, odchytávání výjimek.

7.2 Zadání 5.2 - čtení z klávesnice s pomocí třídy Scanner

Specifikace

Vyzkoušejte si práci s třídou `Scanner`, konkrétně její využití při načítání vstupů z klávesnice. Vytvořte si třídu s atributy `login`, `name` a `age`, které budete nastavovat v konstruktoru a také k nim umožněte přístup pomocí get a set metod. Poslední metodou bude `toString()`, která bude sloužit pro výpis daného studenta.

Druhou třídou bude `StudentAgenda` s atributem scanner typu `Scanner` a kolekcí `students` typu `List<Student>`. Do těchto atributů vytvořte nové instance v konstruktoru. Při vytváření Scanneru předejte jako parametr `System.in`. Třída bude obsahovat pouze jednu veřejnou metodu `execute()`, ve které vytvoříte menu s možnostmi: přidat studenta, změnit studentovo jméno, změnit jeho věk, zobrazit počet studentů, vypsat všechny studenty a ukončit program. Pro jednotlivé možnosti si vytvořte příslušné privátní metody, které budou načítat data z klávesnice pomocí Scanneru a odpovídajících metod. Při načítání čísel ošetřete výjimku `InputMismatchException`, která se vyvolá při zadání hodnoty jiného datového typu.

Testování

Vytvořte instanci třídy `StudentAgenda` a zavolejte metodu `execute()`. Vyzkoušejte, že jednotlivé části programu fungují správným způsobem. Vyzkoušejte také zadání řetězce tam, kde se očekává číslo.

Omezení: při vytváření studenta je nutno zadat všechny jeho atributy.

Obtížnost: 2 - mírně pokročilá.

Zahrnuté oblasti: načítání z klávesnice, Scanner, generické kolekce, odchytávání výjimek.

7.3 Zadání 5.3 - ukládání a čtení textového souboru

Specifikace

Naučte se pracovat s textovým souborem, ukládáním do něj a zpětným načtením informací. Vytvořte abstraktní třídu `Person` s chráněnými atributy `firstName`, `lastName` a `age`. Pro tyto atributy vytvořte get a set metody. Dále naprogramujte třídu `Employee` dědící z `Person`. Třída bude rozšířena o atributy `department` typu `String` a `salary` typu `int`. Dopište potřebné get a set metody. Třída bude mít také dva konstruktory - bezparametrický a parametrický, ve kterém se budou zadávat všechny atributy. Přepište také metodu `toString()`, abyste umožnili výpis daného zaměstnance. Druhou třídou dědící z `Person` bude `Student`. Třída bude obsahovat atributy `year` typu `int` a `startOfStudy` typu `String`. Implementaci třídy vytvořte obdobným způsobem jako u třídy `Employee`.

Poslední třídou bude `PersonalAgenda`, která bude obsahovat generickou kolekci osob, metodu pro přidání osoby do kolekce a metodu pro vypsání všech osob z kolekce. Další metodou bude `saveToFile()`, která uloží celou kolekci do textového souboru *persons.txt*. Ukládejte vždy na jeden řádek jeden atribut. Nezapomeňte, že v kolekci jsou instance tříd `Student` a `Employee` a pro jejich zpětné načtení je nutno si uložit i o jakou instanci se jedná. Poslední metodou bude `loadFromFile()`, která vymaže kolekci osob a poté ji naplní záznamy přečtenými z textového souboru. Pro zápis a čtení souboru použijte třídy `FileWriter` a `FileReader` obalené bufferovacími třídami. Po ukončení práce s danými proudy je nezapomeňte uzavřít.

Testování

Vytvořte si novou agendu a uložte do ní několik studentů a zaměstnanců, poté si nechte celou kolekci vypsat a uložit do souboru. Můžete se přesvědčit, že se vám na disku opravdu vytvořil textový soubor. Nyní dosavadní kód zakomentujte a vytvořte novou instanci agendy, načtěte data ze souboru a vypište kolekci. Porovnejte, zda výpis odpovídá uloženým datům.

Omezení: cesta a název souboru jsou zadány přímo v daných metodách.

Obtížnost: 3 - pokročilá.

Zahrnuté oblasti: zápis a čtení textového souboru, generické kolekce, odchytávání výjimek, dědičnost.

7.4 Zadání 5.4 - ukládání a čtení binárního souboru

Specifikace

Vyzkoušejte si práci s ukládáním do binárního souboru a následným načtením uložených informací. Vytvořte si třídu `Task` a atributy `task` typu `String`, `priority` typu `int` a `finished` typu `boolean`. Napište bezparametrický a parametrický konstruktor a také get a set metody. Nakonec ještě přepište metodu `toString()`.

Druhou třídou bude `TaskList` s generickou kolekcí `tasks` typu `List<Task>`. Pro práci s kolekcí napište metody pro vkládání a výpis všech úkolů z kolekce. Naprogramujte zde metodu `saveToBinaryFile()`, která bude ukládat atributy všech prvků do souboru *tasks.bin*. Pro ukládání použijte třídu `FileOutputStream`, kterou předáte třídě `DataOutputStream`. Při ukládání musíte použít metody v závislosti na tom, jaký datový typ chcete uložit. Poslední metodou bude `loadFromBinaryFile()`, kde uložené hodnoty načtete zpátky do kolekce. Použijte třídy `DataInputStream` a `FileInputStream`. Nezapomeňte nakonec uzavřít všechny datové proudy.

Testování

Vytvořte si instanci třídy `TaskList` a uložte do ní několik úkolů, ty si poté nechte vypsat a uložit do souboru. Nyní si vytvořte novou instanci `TaskList`, do které načtete hodnoty ze souboru. Kolekci si vypište a porovnejte s kolekcí, kterou jste ukládali.

Omezení: cesta a název souboru jsou zadány přímo v daných metodách.

Obtížnost: 3 - pokročilá.

Zahrnuté oblasti: zápis a čtení binárního souboru, generické kolekce, odchytávání výjimek.

7.5 Zadání 5.5 - serializace

Specifikace

Vytvořte třídu `Item` reprezentující položku v obchodě. Třída bude mít atributy `type`, `name` a `price`. Hodnoty nastavte v konstruktoru a také vytvořte get a set metody a přepište metodu `toString()`. Tato třída bude implementovat rozhraní `Serializable`.

Nyní si vytvořte abstraktní třídu `Shop` s chráněným atributem typu `List<Item>` a metodami pro přidání a výpis prvků kolekce. Dále bude třída obsahovat dvě abstraktní metody `serializeToFile()` a `deserializeFromFile()`. Poslední třídou bude `Bakery` dědící ze třídy `Shop`. V konsktruktoru vytvořte instanci `ArrayList<Item>`, kterou uložíte do zděděného atributu. Naimplementujte zděděné abstraktní třídy. Data ukládejte do souboru *bakery.dat*. Pro serializaci použijte třídu `ObjectOutputStream`, které předáte `FileOutputStream`. Pro deserializaci použijte `ObjectInputStream` a `FileInputStream`. Při načítání metoda vrací objekt typu `Object`, a proto je nutné jej přetypovat. V obou metodách odchyťte potřebné výjimky a vždy uzavřete použité proudy.

Testování

Vytvořte si novou pekárnu a vložte do ní několik objektů typu `Item`. Prvky si nechte napřed vypsat a poté serializovat do souboru. Nyní si vytvořte druhou pekárnu a nechte do ní prvky deserializovat a následně vypsat. Zjistěte, zda se všechny vámi uložené prvky správně načetly.

Omezení: cesta a název souboru jsou zadány přímo v daných metodách.

Obtížnost: 3 - pokročilá.

Zahrnuté oblasti: serializace a deserializace, generické kolekce, odchytávání výjimek, dědičnost.

7.6 Zadání 5.6 - práce s GZIP

Specifikace

Naprogramujte jídelní menu, které budete ukládat a načítat z komprimovaného souboru. Vytvořte si metodu `Menu` se čtyřmi privátními atributy typu `List<String>`, budou to: `salads`, `mainCourses`, `deserts` a `drinks`. V konstruktoru do těchto atributů uložte nové instance `ArrayList<String>`. Pro každý seznam vytvořte metodu pro přidání prvku a poté jednu metodu, která vypíše postupně všechny seznamy. Další metodou bude `saveToFile()`, která uloží postupně celé menu do souboru *menu.gz*. Pro správné zpětné načtení dat musíte identifikovat, kde končí jednotlivé seznamy a začínají další. Každý prvek seznamu ukládejte na samostatný řádek. Pro uložení použijte `BufferedWriter`, kterému předáte `OutputStreamWriter`, který bude mít v konstruktoru zadaný `GZIPOutputStream`, a ten bude mít předaný `FileOutputStream`. Nakonec napište metodu `loadFromFile()`, ve které načtete data ze souboru do jednotlivých kolekcí, nezapomeňte správně rozlišit jednotlivé kolekce. Pro vytvoření načítacího proudu postupně použijte třídy `BufferedReader`, `InputStreamReader`, `GZIPInputStream` a `FileInputStream`. V obou metodách ošetřete potřebné výjimky a uzavřete proudy.

Testování

Vytvořte si nové `Menu`, do kterého přidejte několik jídel do každé kategorie. Jídelní lístek si nechte celý vypsat a následně uložit do souboru. Teď si vytvořte druhou instanci `Menu`, lístek načtěte ze souboru, vypište a ověřte, zda je vše v pořádku.

Omezení: cesta a název souboru jsou zadány přímo v daných metodách.

Obtížnost: 3 - pokročilá.

Zahrnuté oblasti: zápis a čtení komprimovaného souboru, generické kolekce, odchytávání výjimek.

7.7 Zadání 5.7 - práce s XML

Specifikace

Naprogramujte knihovnu, kterou budete ukládat do XML a následně ji z XML načítat. Vytvořte třídu `Book` s atributy `id`, `author`, `title` a `genre`, bezparametrickým a parametrickým konstruktorem, get a set metodami a metodou `toString()`. Druhou třídou bude `Bookstore`

s privátní kolekcí knih typu `List<Book>`. Vytvořte metody pro vložení knihy a pro vypsání všech knih. Dále vytvořte metody pro uložení do XML a pro načtení z XML. Data ukládejte do souboru *bookstore.xml*. Metody pro práci s XML můžou vyvolat některé výjimky, proto je podle potřeby odchytávejte. Ukázka práce s XML viz. Příloha D.

Testování

Vytvořte novou knihovnu, do které vložíte několik knížek. Knížky si nechte pro kontrolu vypsat a poté je uložte do XML. Nyní si vytvořte druhou instanci knihovny a knížky z XML načtěte a vypište. Porovnejte oba dva výpisy a zkontrolujte, že jsou stejné. Nakonec si najděte daný soubor na disku, otevřete ho a zkontrolujte, zda má správnou XML strukturu.

Omezení: cesta a název souboru jsou zadány přímo v daných metodách.

Obtížnost: 4 - velmi pokročilá.

Zahrnuté oblasti: zápis a čtení XML souboru, generické kolekce, odchytávání výjimek.

7.8 Zadání 5.8 - práce s PrintWriter a třídou File

Specifikace

Naprogramujte aplikaci, která umožní zobrazit a uložit obsah dané složky na disku do souboru a následně soubor přečíst. Program bude také celý ovládaný pomocí vstupů z klávesnice. Vytvořte si třídu `FileLister` s atributem `path` typu `String`. Napište metodu `execute()`, která bude obsahovat menu programu s možnostmi: nastavit cestu, zobrazit aktuální cestu, uložit obsah složky z dané cesty do souboru, zobrazit obsah dané složky na konzoli, přečíst informace ze souboru a ukončit program. Pro jednotlivé možnosti si naprogramujte odpovídající metody. Pro zjištění obsahu složky použijte třídu `File` a její metody `list()` a `listFiles()`. U výpisu na konzoli zobrazte pouze seznam souboru a složek, při uložení do souboru přidejte informaci, zda se jedná o soubor či složku a u souboru také o jeho velikost. Pro zápis do souboru použijte třídu `PrintWriter`, které předáte instanci třídy `FileWriter`. Pro načítání můžete použít např. `BufferedReader` a `FileReader`. Způsob, jakým budete načítat vstupy z klávesnice, záleží na vás. Nezapomeňte ošetřit potřebné výjimky a všechny proudy korektně zavřít.

Testování

Vytvořte si instanci třídy `FileLister` a zavolejte na ni metodu `execute()`. Vyzkoušejte všechny možnosti programu, zkontrolujte, že výpis složky programem odpovídá skutečné složce na disku. Po uložení výpisu do souboru ho následně přečtěte v programu a také si ho otevřete na disku a zkontrolujte.

Omezení: cesta a název souboru pro uložení jsou zadány přímo v daných metodách.

Obtížnost: 4 - velmi pokročilá.

Zahrnuté oblasti: PrintWriter, File, načítání z klávesnice, odchytávání výjimek.

7.9 Zadání 5.9 - komunikace pomocí socketů

Specifikace

Vytvořte Client-Server aplikaci, ve které bude klient posílat zprávy serveru a ten zprávy vypíše na konzoli velkými písmeny a zároveň uloží do souboru malými písmeny. Komunikace probíhá do doby než je zadáno "end".

Vytvořte si třídu `Server`, která bude obsahovat spouštěcí metodu `main()`. Třída bude obsahovat atribut `port` typu `int`, který budete zadávat v konstruktoru a jednu metodu `work()`. V metodě vytvořte instanci `ServerSocket` na daném portu a poté na ni zavolejte metodu `accept()`, která vám vrátí objekt typu `Socket`. Pro čtení ze socketu si vytvořte `BufferedReader`, kterému předáte `InputStreamReader` a tomu předáte stream, který vrací metoda `getInputStream()` třídy `Socket`. Pro zápis do souboru použijte `BufferedWriter` a `FileWriter`. Správně odchyťte výjimky a proudy včetně socketu uzavřete. V metodě `main()` vytvořte `Server` na daném portu, např. 5000 a zavolejte metodu `work()`.

Druhá třída `Client` bude také obsahovat metodu `main()`. Třída bude mít stejný atribut `port` a metodu `work()`. V této metodě vytvořte instanci `Socket`, které předáte IP adresu a zadaný port. V našem případě bude IP adresou tzv. loopback, čili 127.0.0.1. Pro posílání zpráv serveru použijte `PrintWriter`, kterému předáte `OutputStreamWriter` a tomu předáte výstupní stream pomocí metody `getOutputStream()` třídy `Socket`. Po předání zprávy proudu je třeba zavolat metodu `flush()`, aby se zpráva okamžitě odeslala. Pro načítání z klávesnice použijte vámi zvolený způsob. Nezapomeňte ošetřit potřebné výjimky a otevřené proudy a sockety zavřít. V metodě `main()` si vytvořte instanci třídy `Client` se stejným portem, jaký jste zadali na serveru a zavolejte metodu `work()`.

Testování

Spusťte si `Server` a následně `Client`. Spuštění každého programu proveďte z příkazového řádku ve zvláštním okně. Spuštění provedete příkazem *java Server* nebo *java Client* ve složce, kde se nachází soubory *.class*. Z klienta pošlete několik zpráv a zkontrolujte výpisy na serveru, poté zadejte příkaz *end* a ověřte, že se server i klient ukončily. Následně si otevřete soubor vytvořený serverem a zkontrolujte jeho obsah.

Omezení: cesta a název souboru pro uložení jsou zadány přímo v dané metodě; port se zadává v konstruktoru a musí být stejný; nejdříve je potřeba spustit server.

Obtížnost: 4 - velmi pokročilá.

Zahrnuté oblasti: Client-Server, Socket, PrintWriter, ukládání do textového souboru, načítání z klávesnice, odchytávání výjimek.

7.10 Zadání 5.10 - kvíz

Specifikace

Naprogramujte kvíz, který si načte otázky z XML a poté bude ovládaný pomocí vstupů z klávesnice. Po skončení kvízu se vygeneruje hodnocení, které se uloží do textového souboru. Vytvořte si třídu `Item`, která reprezentuje jednu otázku kvízu. Třída bude mít atributy: `question`, `option1`, `option2` a `option3` - všechny typu `String` a dále atributy `rightAnswer`, značící číslo správné odpovědi a `answer`, uchovávající číslo odpovědi uživatele. Vytvořte bezparametrický a parametrický konstruktor a get a set metody.

Druhou třídou bude `Quiz` s atributy `allQuestions` a `generatedQuestions` typu `List<Item>`. V konstruktoru vytvořte odpovídající instance a zavolejte privátní metodu `loadFromXML()`, která načte otázky z XML souboru do seznamu `allQuestions`. Buď si vytvořte vlastní soubor s otázkami nebo použijte soubor z příloh. Další privátní metody budou zajišťovat náhodné vygenerování tří otázek, které se uloží do seznamu `generatedQuestions`. Zajistěte, aby se nevygenerovala otázka, která již v seznamu je. Poslední privátní metodou bude `saveResultToFile()`, ve které uložíte informace o daném kvízu do souboru, uložte také datum a čas, kdy byl kvíz vykonán. Informace přidávejte na konec textového souboru - nevytvářejte po každé nový soubor. Nakonec bude mít třída jedinou veřejnou metodu `startQuiz()`, která se spustí pouze tehdy, pokud jsou v XML souboru alespoň tři otázky. V této metodě zavolejte metodu pro vygenerování otázek a poté je postupně zobrazujte na konzoli, ukládejte zadané odpovědi a hned vypisujte, zda je odpověď správná nebo ne. Nakonec zavolejte metodu pro uložení výsledku do souboru.

Testování

Vytvořte si instanci třídy `Quiz` a zavolejte metodu `startQuiz()`. Odpovězte na všechny otázky, odpovězte dobře i špatně. Nyní se podívejte na disk a zkontrolujte soubor s hodnocením, soubor zavřete a spusťte kvíz znovu. Po jeho skončení znovu otevřete soubor a ověřte, že se informace do souboru přidaly.

Omezení: cesta a název souboru pro hodnocení jsou zadány přímo v dané metodě; kvíz lze spustit jen, když existují alespoň 3 otázky; jeden kvíz má 3 otázky.

Obtížnost: 5 - těžká.

Zahrnuté oblasti: načítání XML souboru, ukládání do textového souboru, načítání z klávesnice, odchytávání výjimek.

8 Paralelismus a souběžná vlákna

8.1 Zadání 6.1 - vlákna pomocí anonymní třídy

Specifikace

Naprogramujte čítač, který poběží v samostatném vlákně. Vytvořte si třídu `Timer` s atributy `tick` a `delay` datového typu `int` a poté atributy `end` a `paused` typu `boolean` a nakonec atribut `thread` typu `Thread`. Třída bude obsahovat parametrický konstruktor s parametrem `delay` a get a set metody pro atributy `tick`, `end` a `delay`. Další metodou bude `increase()`, která zvedne tick o 1. Metoda `start()` vytvoří do atributu `thread` novou instanci třídy `Thread`, které předáte v konstruktoru novou instanci `Runnable`, kterou nadefinujete jako anonymní třídu. Ta bude obsahovat pouze jednu metodu a to metodu `run()`, ve které v cyklu, dokud nebude atribut `end` roven *true*, vlákno uspíte na dobu `delay` a poté zavoláte metodu `increase()`. Jako poslední příkaz metody `start()` bude spuštění vlákna pomocí metody `start()` na atribut `thread`. Posledními metodami budou `stop()`, `pause()` a `continueCount()`, které umožní zastavování čítače, zároveň nedovolte spustit čítač pokud již běží.

Testování

Vytvořte si nový čítač se zpožděním 100. Čítač spusťte a hlavní vlákno uspěte na 1 sekundu, poté čítač zastavte a počkejte 2 sekundy. Následně čítač spusťte a nechejte běžet půl sekundy. Zkontrolujte, že vám čítač počítal znovu od 1. Nyní čítač pozastavte metodou `pause()`, počkejte další půl sekundy a pokračujte v počítání. Zkontrolujte si, že čítač pokračuje opravdu dál od místa, kde skončil. Nechejte čítač chvíli počítat a poté ho zastavte, opět chvíli počkejte a změňte zpoždění na jednu sekundu a čítač spusťte od začátku. Po několika sekundách zkuste zavolat metodu `continueCount()` a ověřte, že se čítač znovu nespustí. Nakonec už jen čítač zastavte.

Omezení: při vytváření čítače je nutno zadat zpoždění; v rámci jedné instance třídy `Timer` může běžet jenom jeden čítač.

Obtížnost: 2 - mírně pokročilá.

Zahrnuté oblasti: vlákna, Thread, Runnable, anonymní třídy, odchytávání výjimek.

8.2 Zadání 6.2 - vlákna pomocí dědění z třídy Thread

Specifikace

Vytvořte simulaci výrobních linek běžících paralelně. Naprogramujte třídu `Truck` s atributy `String product`, `int maxWeight`, `int currentWeight` a `boolean full`. Konstruktor bude přijímat parametry `product` a `maxWeight`. Třída bude obsahovat jedinou metodu `load(int weight)`, která bude vracet *true*, pokud se podaří naložit produkt a *false*, pokud je `Truck` plný.

Druhou třídou bude `ProductionLine`, která bude dědit z `Thread`. Třída bude mít atribut `truck` typu `Truck` a poté atributy typu `int`: `timeToFinish` určující potřebný čas k vyrobení jednoho výrobku, `timeOfWork` definující celkovou dobu práce výrobní linky, `weight` určující

váhu jednoho výrobku, a atributy `currentTime`, `numberOfMadeProducts` a `numberOfLoadedProducts`. V konstruktoru přijímejte parametry `timeToFinish`, `timeOfWork`, `truck`, `weight` a parametr `name`, který předáte konstruktoru předka. Přepište metodu `toString()`, která bude vracet výsledek metody `getName()`, která je nadefinovaná v předkovi a dále naprogramujte metodu `printStatus()`, která vypíše kolik a jakých produktů se vyrobilo a naložilo za jakou dobu. Nakonec přepište metodu `run()` nutnou pro spuštění vlákna. V této metodě budete vyrábět produkty v cyklu od 0 do `timeOfWork`. Když bude výrobek hotov, tak se ho pokusíte naložit. Na konci metody `run()` zavolejte metodu `printStatus()`.

Testování

Vytvořte si tři Trucky s nosností 1500 pro tři druhy výrobků, např. televize, mobily a rádia. Dále si vytvořte tři výrobní linky pro dané výrobky. Televizím nastavte čas pro vyrobení 20, celkový čas práce 200 a váhu 20. Mobilům čas výroby 10, celkový čas 2000 a váhu 2. Rádiím čas výroby 5, celkový čas 4000 a váhu 5. Nyní už jen spusťte jednotlivé vlákna (výrobní linky) pomocí metody `start()`. Pokud jste nastavili výše zadané hodnoty, tak by se vám mělo vyrobit 100 televizí a z toho 75 naložit. Dále 800 rádií a 300 naložit a nakonec 200 mobilů a všechny naložit.

Omezení: při vytváření Trucků a výrobních linek je nutno zadat odpovídající parametry.

Obtížnost: 2 - mírně pokročilá.

Zahrnuté oblasti: vlákna, dědičnost.

8.3 Zadání 6.3 - vlákna pomocí rozhraní Runnable

Specifikace

Naprogramujte simulaci pokladen pomocí vláken. Vytvořte si třídu `Customer` s atributy `name` a `time`. `Time` je typu `int` a určuje potřebný čas zákazníka na obsloužení. Hodnoty nastavte v parametrickém konstruktoru a také vytvořte get a set metody.

Další třída bude abstraktní a bude to `AbstractCashDesk` s chráněnými atributy `id` a generickou frontou `customers`. Vytvořte metodu pro přidání zákazníka do fronty a abstraktní metodu `serveCustomer()`. Z této třídy budou dědit `CashDesk` a `ExpressCashDesk`. Tyto třídy budou implementovat rozhraní `Runnable` a jejich implementace bude stejná, pouze s tím rozdílem, že expresní pokladně bude trvat obsloužení zákazníka poloviční dobu. V konstruktoru těchto tříd nastavte atribut `id`. Doimplementujte metodu `serverCustomer()`, která vyjme zákazníka z fronty, na konzoli vypíše, že ho obsluhuje a poté se dané vlákno uspí na čas zadaný v objektu zákazníka (u expresní pokladny tento čas vydělte dvěma). Dále musíte doplnit metodu `run()`, která bude v cyklu, dokud nebude fronta prázdná, volat metodu `serveCustomer()`. Nakonec ještě napište metodu `start()` pro spuštění. V té vytvoříte novou instanci `Thread` s parametrem `this` a zavoláte na ni metodu `start()`.

Testování

Vytvořte si dvě normální pokladny a jednu expresní. Do fronty každé pokladny vytvořte a vložte 5 až 10 zákazníků. Čas u zákazníků nastavujte pomocí třídy `Random` z rozmezí 100 - 500. Všechny tři pokladny spusťte a sledujte jednotlivé výpisy. Ověřte, že pokladny opravdu běží paralelně.

Omezení: při vytváření zákazníků a pokladen je nutno zadat odpovídající parametry.

Obtížnost: 2 - mírně pokročilá.

Zahrnuté oblasti: vlákna, dědičnost, Runnable, generické kolekce, odchytávání výjimek.

8.4 Zadání 6.4 - paralelní běh pomocí vláken

Specifikace

Naprogramujte simulaci závodu aut. Vytvořte si třídu `Car`, která bude umožňovat vláknové spuštění. Třída bude obsahovat atributy `acceleration`, `maxSpeed`, `currentSpeed` a `distancePassed` typu `double` a atribut `time` typu `long`. V parametrickém konstruktoru nastavte `acceleration` a `maxSpeed`. Dále vytvořte get metodu pro atribut `time`, kde vrátíte `time` převedený z nanosekund na sekundy a přepište metodu `toString()`, která bude vracet `currentSpeed` a `distancePassed`. V metodě `run()` si na začátku uložte do proměnné aktuální čas pomocí statické metody `System.nanoTime()`, poté v cyklu, dokud nebude `currentSpeed` rovna `maxSpeed`, zvyšujte vždy po 100 milisekundách `currentSpeed` o `acceleration` a zároveň zvyšujte `distancePassed` o hodnotu `currentSpeed/60`. Následně napište další cyklus, který bude probíhat, dokud nebude uražená vzdálenost větší než 100. V něm pouze zvyšujte hodnotu `distancePassed` opět o `currentSpeed/60`. Na konci metody `run()` zjistěte aktuální čas a rozdíl časů z konce a začátku uložte do atributu `time`.

Testování

Vytvořte si čtyři auta. První se zrychlením 20 a maximální rychlostí 250, další se zrychlením 30 a rychlostí 220, třetí auto bude mít zrychlení 15 a rychlost 190 a poslední bude mít zrychlení 25 a maximální rychlost 240. Všechna vlákna spusťte a v cyklu, dokud všechna vlákna běží, je co 75 milisekund vypisujte. Po skončení vypište cílový stav všech aut a čas, jak dlouho vlákno běželo. Pokud jste zadali výše uvedené hodnoty, tak by ujetá vzdálenost jednotlivých aut měla být: 101, 102, 101,83 a 102,75. Všimněte si, že čas běhu vlákna se lehce liší při každém spuštění.

Omezení: při vytváření aut je nutno zadat zrychlení a maximální rychlost.

Obtížnost: 3 - pokročilá.

Zahrnuté oblasti: vlákna, odchytávání výjimek.

8.5 Zadání 6.5 - synchronizace vláken

Specifikace

Vyzkoušejte si typický příklad typu producent/konzument. Vytvořte si třídu `SharedData` s generickou kolekcí `data` typu `List<String>` a atribut `size`. Ten nastavte v parametrickém konstruktoru. Třída bude obsahovat metody `insertItem(String item)` a `getItem()`, která bude vracet `String`. Obě dvě metody musí být označeny jako `synchronized`, aby k metodě přistupovalo vždy jen jedno vlákno. V metodě pro vkládání nejdříve zjistěte, jestli je kolekce plná. V případě, že ano, vypište hlášku, že Producent čeká a následně zavolejte metodu `wait()`, která se dědí z třídy `Object` a která pozastaví provádění vlákna, dokud nedostane signál k pokračování. Následně vypište, že Producent již může zapisovat a vložte data. Nakonec zavolejte metodu `notifyAll()`, která se také dědí z `Object` a která uvolní všechna pozastavená vlákna. V metodě pro výběr objektu ověřte, zda je kolekce prázdná a opět vypište, že Konzument čeká na data a zavolejte metodu `wait()`, následně vypište, že již může pokračovat, data vyjměte a zavolejte `notifyAll()`. Nakonec už jen data vraťte.

Třída `Consumer` bude podporovat vláknové spuštění a bude mít atribut `data` typu `SharedData` a `name` typu `String`. Tyto hodnoty nastavte v konstruktoru. V metodě `run()` v nekonečném cyklu vyjměte data a vlákno uspěte na dobu vygenerovanou pomocí třídy `Random` v rozmezí 100 - 1000 milisekund.

Třída `Producer` bude mít obdobnou implementaci jako konzument. Navíc bude obsahovat atribut `numberOfProducts`, který bude udávat kolik produktů má vygenerovat. V metodě `run()` vygenerujte a vložte v cyklu daný počet produktů. Produkty generujte jako jméno producenta s číslem daného produktu. Mezi jednotlivým vkládáním opět uspěte vlákno na dobu 100 - 1000 milisekund.

Testování

V metodě `main()` si vytvořte sdílená data o velikosti 10. Dále si vytvořte 4 producenty, kteří budou generovat několik záznamů (10 - 20) a 2 konzumenty. Všechna vlákna spusťte a sledujte výpisy, ve kterých byste měli vidět, že vlákna na sebe čekají.

Omezení: při vytváření producenta je nutno zadat jeho název a počet produktů; při vytváření konzumenta je nutno zadat jeho název; konzumenti běží do nekonečna, a proto je potřeba program manuálně ukončit.

Obtížnost: 3 - pokročilá.

Zahrnuté oblasti: vlákna, synchronizované metody, metody wait() a notifyAll(), generické kolekce, odchytávání výjimek.

8.6 Zadání 6.6 - paralelní stahování webových stránek

Specifikace

Vytvořte si třídu `Page` s atributy `url` a `page`. V konstruktoru nastavte `url`. Vytvořte zde get a set metody a přepište metodu `toString()`. Vytvořte další třídu `Downloader` podporující

vláknové spuštění. Třída má jeden atribut `page` typu `Page`, který nastavíte v konstruktoru a vytvoříte pro něho get metodu. V metodě `run()` stáhněte danou stránku. Pro stažení si vytvořte instanci třídy `URL`, které předáte adresu stránky a poté použijte `DataInputStream`, kterému předáte `BufferedInputStream` a tomu předejte stream z `url` pomocí metody `openStream()`. Stránku přečtěte po řádcích a výsledek uložte do atributu `page`.

Poslední třídou bude `DownloadManager`, která bude mít dvě fronty `requests` a `results` typu `Queue<Page>`. Dále bude obsahovat seznam `downloaders` typu `List<Downloader>` a atribut `numberOfThreads`, který nastavíte v konstruktoru. Napište metody pro přidání stránky do fronty požadavků a pro zobrazení všech výsledků. Poslední metodou bude `download()`, ve které vytvoříte zadaný počet vláken a stáhnete všechny stránky.

Testování

Vytvořte si `DownloadManager` a zadejte např. dvě vlákna. Vložte požadavky na alespoň 3 stránky a nechte je stáhnout. Poté si zobrazte výsledky a zkontrolujte, zda se vám správně stáhly.

Omezení: při vytváření `DownloadManager` je nutno zadat počet vláken; při vytváření stránky je nutno zadat její url.

Obtížnost: 3 - pokročilá.

Zahrnuté oblasti: vlákna, datové proudy, generické kolekce, odchytávání výjimek.

8.7 Zadání 6.7 - čtení souboru ve vláknech

Specifikace

Vaším úkolem je na pozadí přečíst textový soubor s čísly a daná čísla sečíst. V hlavním vlákně budete vypisovat průběh. Jako první si napište třídu, která vám vygeneruje textový soubor s celými čísly v rozmezí 0 - 100. Ukládejte vždy jedno číslo na řádek a vygenerujte milión nebo 10 miliónů řádků v závislosti na výkonnosti vašeho počítače.

Nyní si vytvořte třídu `Reader`, která bude podporovat vláknové spuštění a bude mít atributy `sum` a `finished`. Pro oba atributy vytvořte get metody. V metodě `run()` postupně načítejte soubor po řádcích a čísla sčítejte a ukládejte do atributu sum. Vždy po přičtení jednoho čísla zavolejte statickou metodu `Thread.yield()`, kterou předáte řízení dalšímu vláknu.

Poslední třídou bude `Watcher`, která bude mít jediný atribut typu `Reader` a bude také podporovat vláknové spuštění. Atribut `reader` nastavte v konstruktoru. V metodě `run()` vytvořte smyčku, která poběží dokud nebude načtený celý soubor. V této smyčce vypisujte každých 100 milisekund průběžný součet čísel. Na konci vypište finální součet.

Testování

V metodě `main()` si nejprve vygenerujte textový soubor, s kterým budete dále pracovat. Vytvořte si instance `Reader` a `Watcher` a obě vlákna spusťte. Nyní už jen sledujte výpisy, jak se postupně soubor načítá. Postupné výpisy jsou určeny pouze pro zpětnou vazbu pro uživatele, že se něco na pozadí opravdu děje.

Omezení: nejdříve je potřeba vytvořit `Reader` a potom `Watcher`, abyste mohli `Reader` předat; název a cesta souboru je zadána přímo v dané metodě.

Obtížnost: 3 - pokročilá.

Zahrnuté oblasti: vlákna, Thread, datové proudy, odchytávání výjimek.

8.8 Zadání 6.8 - seřazení pole pomocí vláken

Specifikace

Seřaďte pole celých čísel bez pomoci vláken a potom s pomocí vláken. U obou variant změřte čas a porovnejte ho. Pro seřazení použijte např. metodu select sort. Vytvořte si tedy třídu `SelectSort`, která bude umožňovat vláknové spuštění a bude obsahovat atributy `data` a `sortedData` typu `int[]` a atributy `from` a `to` typu `int`, které budou určovat jaká část pole se má seřadit. Vstupní pole a indexy `from` a `to` nastavte v konstruktoru. Vytvořte si také set metodu pro pole seřazených dat. V metodě `run()` naimplementujte seřazení zvolenou metodou tak, aby se seřadila zadaná část pole a překopírovala do pole `sortedData`.

Druhou třídou bude `SortManager` s atributy `data` typu `int[]`, `size` typu `int` a `time` typu `double`. V konstruktoru nastavte parametr `size`. Pro atribut `time` si vytvořte get metodu. Další metodou bude `generate()`, která vygeneruje čísla v rozmezí 0 - 10000 do pole `data` o zadané velikosti `size`. Připravte si také metodu `print()`, která vypíše všechny prvky pole. Nyní si vytvořte metodu `sortWithoutThreads()`, která vytvoří instanci `SelectSort`, předá ji celé pole a pole seřadí. Po seřazení si vložte seřazené data do atributu `data`. Před začátkem seřazování a po jeho dokončení si zaznamenejte čas pomocí `System.nanoTime()` a poté spočtený čas převedený na milisekundy uložte do atributu `time`. Druhou metodou pro seřazení bude `sortWithThreads()`. V té si vytvořte dvě vlákna a každému předejte polovinu pole. Po skončení seřazení si vraťte z každého vlákna seřazené pole do lokálních proměnných. Nyní musíte ještě tato pole spojit dohromady do atributu `data`, napište si proto metodu `merge(int[] part1, int[] part2)`. Opět změřte čas trvání seřazení od spuštění vláken po konec metody `merge()`.

Testování

Vytvořte si instanci `SortManager` s velikostí pole 100. Vygenerujte si pole a nechte si ho vypsat, poté ho seřaďte bez vláken a opět nechte vypsat. To samé proveďte i pro seřazení s vlákny. Ověřte, že se vám pole v obou případech správně seřadilo. Nyní spusťte program znovu, ale nastavte počet prvků na 10000 a tentokrát již pole nemusíte vypisovat - víte, že řazení funguje správně a výpis by byl příliš dlouhý. Nechte si ale tentokrát vypsat čas seřazení bez vláken a čas seřazení s vlákny. Oba časy porovnejte.

Omezení: při vytváření `SortManager` je nutno zadat počet prvků pole; seřazení pomocí vláken probíhá pouze ve dvou vláknech.

Obtížnost: 4 - velmi pokročilá.

Zahrnuté oblasti: vlákna, Thread, seřazení pole.

8.9 Zadání 6.9 - chatovací server a klient pomocí vláken

Specifikace

Naprogramujte server, který bude přijímat zprávy od klientů a přeposílat je na všechny ostatní klienty. Server vytvoří pro každého připojeného klienta nové vlákno. Vytvořte si třídu `Server`, která bude obsahovat spouštěcí metodu `main()`. V konstruktoru třídy se bude nastavovat port a třída bude mít taky generický seznam všech svých vláken. K tomu seznamu vytvořte get metodu. Hlavní metodou, kterou budete spouštět v metodě `main()` bude `work()`, kde vytvoříte nový `ServerSocket` a po připojení klienta vytvoříte nové vlákno, kterému předáte `socket` a `this`. Pro serverové vlákno si napište třídu `ServerThread` s atributy typu `Server` a `Socket`, které budete tedy nastavovat v konstruktoru. Dále budete potřebovat `PrintWriter` a `BufferedReader` pro čtení a posílání zpráv. Napište metodu `send(String line, ServerThread from)`, která projde všechna serverová vlákna a do výstupního proudu zapíše `line`. V metodě `run()` čtěte v nekonečné smyčce vstup a volejte metodu `send()`. Když přijde zpráva *"quit"*, tak cyklus a vlákno ukončete.

Další třídou bude `Client`, která bude taky obsahovat spouštěcí metodu `main()`. Třída bude mít atribut `port` nastavovaný v konstruktoru a jedinou metodu `work()`, kterou spustíte v metodě `main()`. Implementujte metodu `work()` tak, aby vytvořila nový `Socket` a následně si vytvořila dvě klientská vlákna, kterým budete předávat vstupní a výstupní proudy. První vlákno bude pro čtení a bude mít vstupní proud ze socketu a jako výstupní proud konzoli. Druhé vlákno bude zapisovací a vstupní proud bude z klávesnice a výstupní ze socketu. Zapisovací vlákno nastavte jako démona, aby po ukončení neblokovalo program. Vlákna spusťte a čekejte až skončí čtecí vlákno. Pro klientské vlákno si vytvořte třídu `ClientThread`, které budete předávat v konstruktoru vstupní a výstupní proudy. V metodě `run()` pouze v nekonečné smyčce čtěte data ze vstupního proudu a zapisujte do výstupního. Pokud bude na vstupu *null*, tak vlákno ukončete. Ve všech třídách odchyťte požadované výjimky a nezapomeňte uzavřít všechny proudy a sockety.

Testování

Server i klienta provozujte např. na portu 5000. Spuštění musíte provést přes příkazovou řádku. Spusťte si jeden server a alespoň dva klienty. Vyzkoušejte si posílání zpráv z klientů a ověřte, že se zprávy zobrazují na druhém klientovi. Poté klienty ukončete příkazem *"quit"*.

Omezení: port se zadává v konstruktoru a musí být stejný; nejdříve je potřeba spustit server.

Obtížnost: 5 - těžká.

Zahrnuté oblasti: Client-Server, Socket, vlákna, generické kolekce, datové proudy, načítání z klávesnice, odchytávání výjimek.

9 Závěr

Cílem této bakalářské práce bylo vytvořit sadu zadání pro praktickou výuku objektového programování v programovacím jazyce Java. Tato zadání měla být zaměřena na jednotlivé technologie Javy. Vzniklo celkem 54 zadání rozdělených do 6 kategorií. Jednotlivá zadání se zaměřují vždy na různé technologie v dané kategorii.

Na začátku své práce jsem představil základní teoretické znalosti o Javě a o objektově orientovaném programování. Poté následovalo teoretické představení jednotlivých kategorií, do nichž jsou zadání rozdělena. U každého zadání jsem se zaměřil na jeho přesnou strukturu a jasné testování. Každé zadání obsahuje podrobně definovanou specifikaci a popis testování. Další součástí zadání je také omezení, které definuje např. jaké konstruktory se mají v třídách použít, jaké datové typy budou ukládat datové struktury a jiné. Každé zadání má také určenu svou obtížnost na stupnici 1 - 5 a také obsahuje seznam všech zahrnutých oblastí. Pro každé zadání jsem vypracoval ukázkové řešení, které je k naleznutí na přiloženém CD.

Dle mého názoru jsou vytvořená zadání použitelná pro výuku objektového programování a Java technologií jak pro samouky, tak pro výuku lektorem. Vzhledem k tematickému zasazení těchto zadání je lze použít jako pomůcku pro výuku předmětu Programovací jazyky I. bakalářského studia na Vysoké škole báňské.

Díky této bakalářské práci jsem si rozšířil své znalosti technologií Java. Vyzkoušel jsem si také tvorbu zadání pro výuku a tím jsem získal na tuto problematiku pohled i z jiného úhlu než jen z pozice studenta.

Použitá literatura

[1] BLOCH, Joshua. Java efektivně: 57 zásad softwarového experta. 1. vyd. Praha: Grada, 2002, 230 s. ISBN 80-247-0416-1.

[2] HEROUT, Pavel. Java bohatství knihoven. 1. vyd. České Budějovice: Kopp, 2003, 242 s. ISBN 80-7232-209-5.

[3] HEROUT, Pavel. Učebnice jazyka Java. 1. vyd. České Budějovice: Kopp, 2001, 349 s. ISBN 80-7232-115-3.

[4] PECINOVSKÝ, Rudolf. Myslíme objektově v jazyku Java: kompletní učebnice pro začátečníky. 2., aktualiz. a rozš. vyd. Praha: Grada Publishing, 2009, 570 s. ISBN 978-80-247-2653-3.

[5] SCHILDT, Herbert. Java 7: výukový kurz. 1. vyd. Brno: Computer Press, 2012, 664 s. ISBN 978-80-251-3748-2.

[6] Java™ Platform, Standard Edition 7 API Specification. [online]. [cit. 2013-04-20]. Dostupné z: http://docs.oracle.com/javase/7/docs/api/

[7] About the Java Technology. In: The Java™ Tutorials [online]. [cit. 2013-04-20]. Dostupné z: http://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html

[8] Lesson: Interfaces. In: The Java™ Tutorials [online]. [cit. 2013-04-20]. Dostupné z: http://docs.oracle.com/javase/tutorial/collections/interfaces/index.html

[9] The Catch or Specify Requirement. In: The Java™ Tutorials [online]. [cit. 2013-04-20]. Dostupné z: http://docs.oracle.com/javase/tutorial/essential/exceptions/catchOrDeclare.html

[10] Mediator. [online]. [cit. 2013-01-27]. Dostupné z: http://www.dofactory.com/Patterns/PatternMediator.aspx

[11] Java (11) - Kontejnery II. In: Programování v jazyku Java [online]. [cit. 2013-01-30]. Dostupné z: http://www.linuxsoft.cz/article.php?id_article=750

[12] How to create XML file in Java – (DOM Parser). [online]. [cit. 2013-02-18]. Dostupné z: http://www.mkyong.com/java/how-to-create-xml-file-in-java-dom/

[13] How to read XML file in Java – (DOM Parser). [online]. [cit. 2013-02-18]. Dostupné z: http://www.mkyong.com/java/how-to-read-xml-file-in-java-dom-parser/

[14] Síťování v Javě: Chatovací server. [online]. [cit. 2013-03-04]. Dostupné z: http://www.root.cz/clanky/sitovani-v-jave-chatovaci-server/

[15] Generika, iterátor. In: Java pro začátečníky [online]. [cit. 2013-03-08]. Dostupné z: http://www.algoritmy.net/article/30003/Generika-iterator-17

[16] Wildcards. In: The Java™ Tutorials [online]. [cit. 2013-03-12]. Dostupné z: http://www.algoritmy.net/article/30003/Generika-iterator-17

Seznam příloh

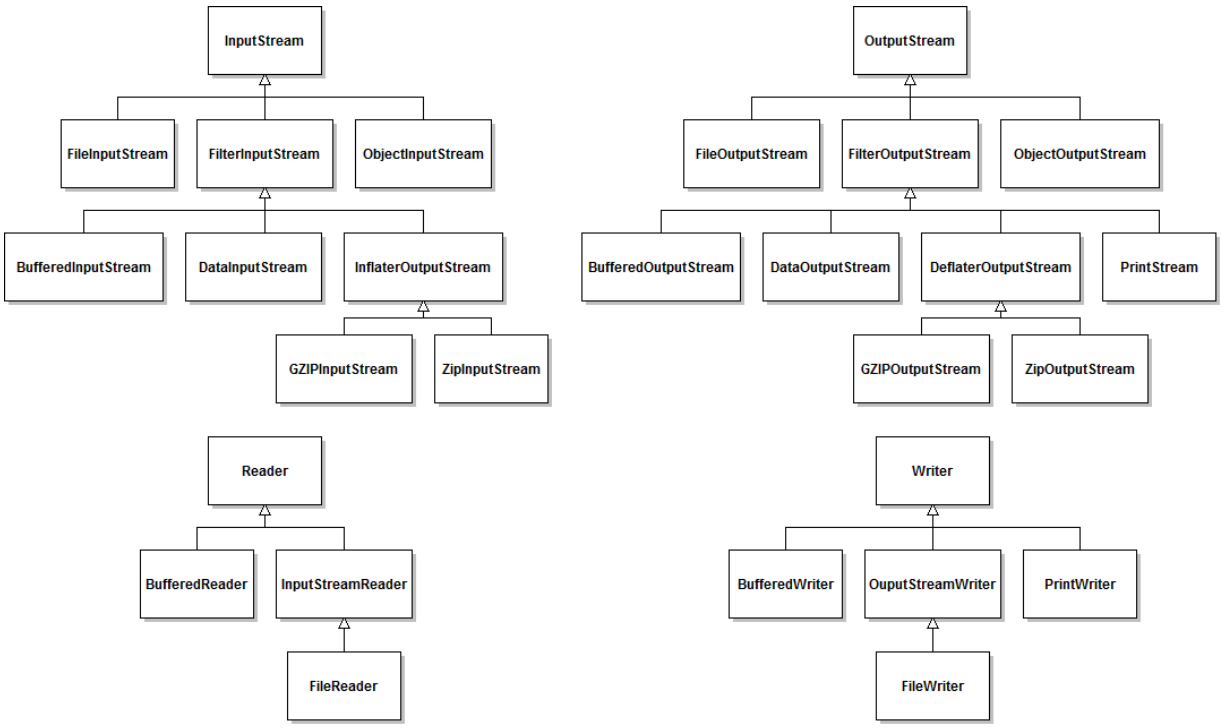
Součástí BP je CD.

Obsah přiloženého CD:

1. Bakalářská práce.docx - vlastní práce
2. Bakalářská práce.pdf - vlastní práce
3. quiz.xml - XML soubor pro zadání 5.10
4. workspace/1_constructors - zadání 1.1
5. workspace/1_events - zadání 1.8
6. workspace/1_inheritance - zadání 1.2
7. workspace/1_innerClasses - zadání 1.6
8. workspace/1_interfaces - zadání 1.3
9. workspace/1_mediator - zadání 1.10
10. workspace/1_object - zadání 1.5
11. workspace/1_packages - zadání 1.4
12. workspace/1_singleton - zadání 1.9
13. workspace/1_treeStructure - zadání 1.7
14. workspace/2_arraylist - zadání 2.5
15. workspace/2_binaryTree - zadání 2.10
16. workspace/2_collections - zadání 2.9
17. workspace/2_hashmap - zadání 2.8
18. workspace/2_hashset - zadání 2.6
19. workspace/2_linkedlist_iterator - zadání 2.4
20. workspace/2_primitiveTypes - zadání 2.3
21. workspace/2_queue - zadání 2.2
22. workspace/2_stack - zadání 2.1
23. workspace/2_treeet - zadání 2.7
24. workspace/3_enummap - zadání 3.5
25. workspace/3_hashmap - zadání 3.4
26. workspace/3_iterator - zadání 3.3
27. workspace/3_queue - zadání 3.6
28. workspace/3_queueGeneric - zadání 3.2
29. workspace/3_restrictions - zadání 3.7
30. workspace/3_stackGeneric - zadání 3.1

31. workspace/3_wildcards - zadání 3.8
32. workspace/4_closeable - zadání 4.7
33. workspace/4_exceptionsAccount - zadání 4.5
34. workspace/4_exceptionsCarSimulation - zadání 4.6
35. workspace/4_exceptionsCoffeeHouse - zadání 4.4
36. workspace/4_exceptionsInsurance - zadání 4.3
37. workspace/4_runtimeExceptions - zadání 4.1
38. workspace/4_runtimeExceptionsWithHandling - zadání 4.2
39. workspace/5_binaryFile - zadání 5.4
40. workspace/5_gzip - zadání 5.6
41. workspace/5_printwriter_file - zadání 5.8
42. workspace/5_quiz - zadání 5.10
43. workspace/5_readingFromKeyboard - zadání 5.1
44. workspace/5_scanner - zadání 5.2
45. workspace/5_serialization - zadání 5.5
46. workspace/5_sockets - zadání 5.9
47. workspace/5_textFile - zadání 5.3
48. workspace/5_xml - zadání 5.7
49. workspace/6_extendedThread - zadání 6.2
50. workspace/6_runnable - zadání 6.3
51. workspace/6_runnableAnonymous - zadání 6.1
52. workspace/6_synchronized - zadání 6.5
53. workspace/6_threadsCarRace - zadání 6.4
54. workspace/6_threadsClientServer - zadání 6.9
55. workspace/6_threadsDownloader - zadání 6.6
56. workspace/6_threadsReadingFile - zadání 6.7
57. workspace/6_threadsSort - zadání 6.8

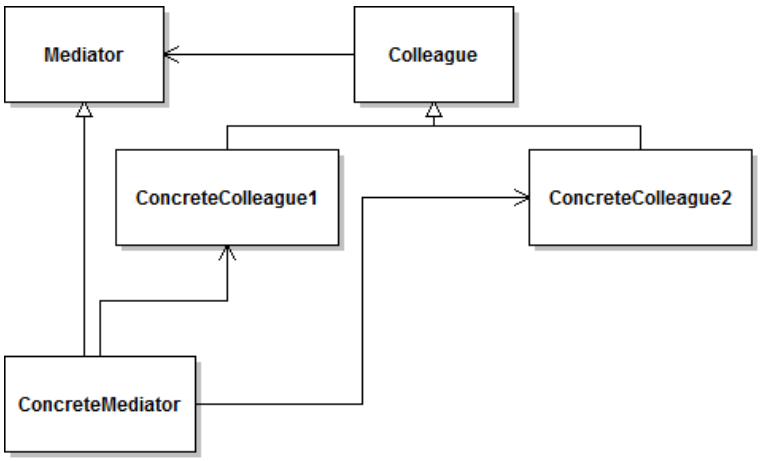
Příloha.A: Vybrané třídy pro práci s datovými proudy



Příloha.B: Ukázka práce s třídou Vector

```
Vector<Node> children = new Vector<Node>();
//přidání
children.add(node);
//procházení
for(int i = 0; i < children.size(); i++) {
      //children.get(i);
}
```

Příloha.C: Třídní diagram návrhového vzoru mediator [10]



Příloha.D: Ukázka práce s XML

```
//vytvoreni
DocumentBuilderFactory docFactory =
DocumentBuilderFactory.newInstance();
DocumentBuilder docBuilder = docFactory.newDocumentBuilder();
Document doc = docBuilder.newDocument();
Element root = doc.createElement("root");
doc.appendChild(root);
Element node = doc.createElement("node");
node.setAttribute("key","value");
node.appendChild(doc.createTextNode("text"));
root.appendChild(node);
//ulozeni
TransformerFactory transFactory = TransformerFactory.newInstance();
Transformer trans = transFactory.newTransformer();
DOMSource source = new DOMSource(doc);
StreamResult result = new StreamResult(new File("file.xml"));
trans.transform(source, result);
//nacteni
Document doc = docBuilder.parse(new File("file.xml"));
NodeList nlist = doc.getElementsByTagName("node");
for (int i=0; i<nlist.getLength(); i++) {
      Node node = nlist.item(i);
      if (node.getNodeType() == Node.ELEMENT_NODE) {
            Element e = (Element) node;
            e.getAttribute("key")));
            e.getTextContent();
      }
}
```

Příloha.E: XML soubor pro zadání 5.10 - kvíz

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<quiz>
  <item>
    <question>Ktery z nasledujicih bloku neexistuje?</question>
    <option1>try-catch</option1>
    <option2>try-finally</option2>
    <option3>catch-finally</option3>
    <rightAnswer>3</rightAnswer>
  </item>
  <item>
    <question>Ktera z nasledujicich trid je korektni?</question>
    <option1>TextReader</option1>
    <option2>Reader</option2>
    <option3>TextWriter</option3>
    <rightAnswer>2</rightAnswer>
  </item>
  <item>
    <question>Kterou metodou se spousti vlakno?</question>
    <option1>run()</option1>
    <option2>start()</option2>
    <option3>execute()</option3>
    <rightAnswer>2</rightAnswer>
  </item>
  <item>
    <question>Jak jsou ulozeny hodnoty v kolekci List?</question>
    <option1>na zaklade indexu</option1>
    <option2>pomoci klic-hodnota</option2>
    <option3>nahodne</option3>
    <rightAnswer>1</rightAnswer>
  </item>
  <item>
```

```
    <question>Muze v Jave trida dedit z vice trid
najednou?</question>
    <option1>ano</option1>
    <option2>ne</option2>
    <option3>zalezi na situaci</option3>
    <rightAnswer>2</rightAnswer>
  </item>
  <item>
    <question>Co znamena zkratka JVM?</question>
    <option1>Java Vital Machine</option1>
    <option2>Java Virtual Monster</option2>
    <option3>Java Virtual Machine</option3>
    <rightAnswer>3</rightAnswer>
  </item>
</quiz>
```